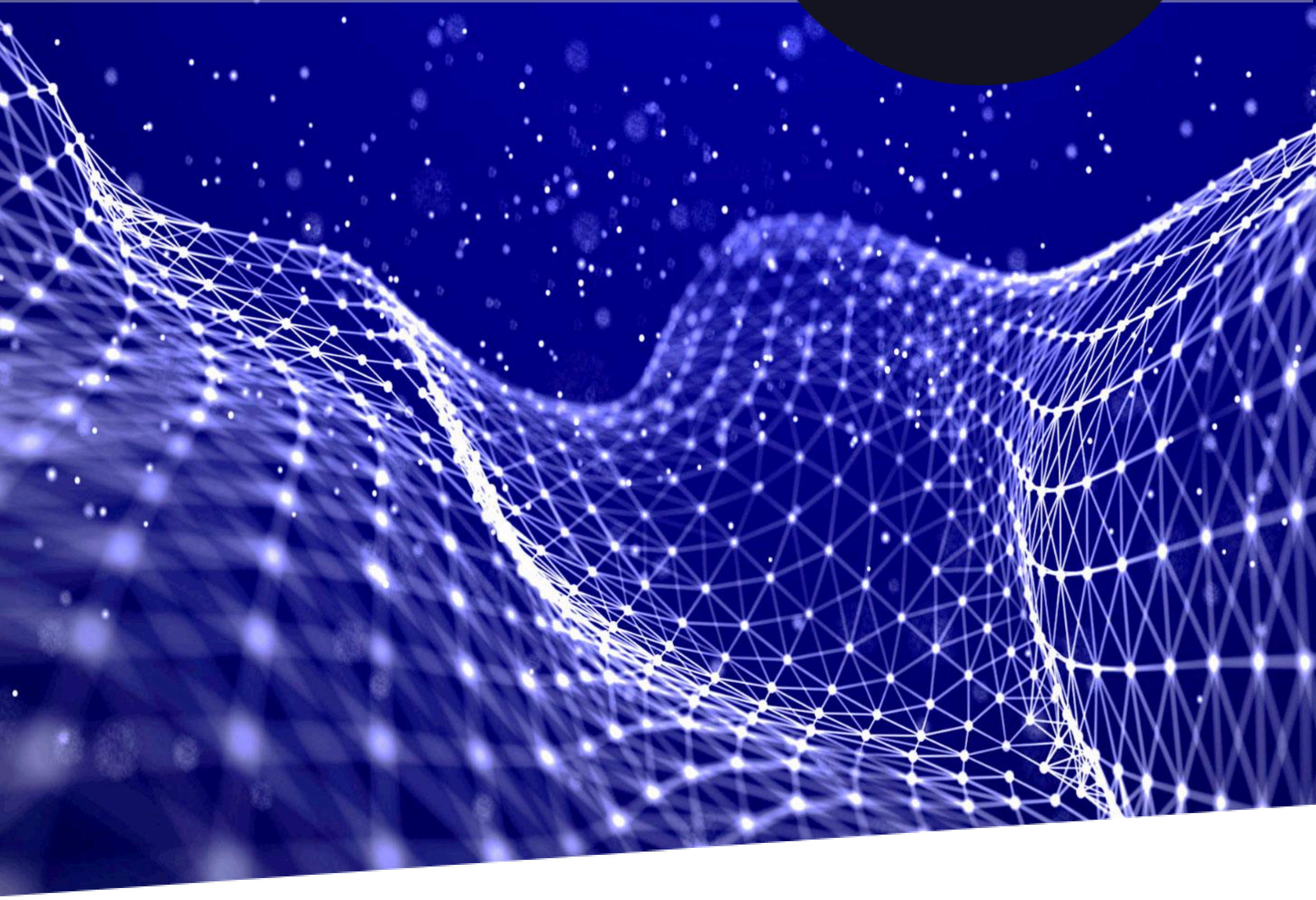
FWX

DeFi Permissionless

Smart Contract Audit Report



Date Issued: 8 Mar 2024

Version: Final v1.1

Public

Table of Contents

Executive Summary

Overview

Valix conducted a smart contract audit to evaluate potential security issues of the **DeFi Permissionless**. This audit report was published on ***8 Mar 2024***. The audit scope is limited to the **DeFi Permissionless.** Our security best practices strongly recommend that the **FWX team** conduct a full security audit for both on-chain and off-chain components of its infrastructure and their interaction. A comprehensive examination has been performed during the audit process utilizing Valix's Formal Verification, Static Analysis, and Manual Review techniques.

About DeFi Permissionless

Permissionless trading listings provide a protocol for openly introducing and accessing FWX's core offerings—specifically its decentralized derivative exchange (DDEX) and lending and borrowing pools (LBPs)—without the need for centralized approvals. This framework ensures that anyone with an internet connection can utilize these features directly, fostering a transparent and borderless environment.

Within the FWX ecosystem, the DDEX relies on the liquidity and real borrowing demand generated by the LBPs. In turn, these LBPs benefit from tangible revenue streams tied to derivative trading orders on the DDEX. The platform's NFT memberships further support these features, enhancing the overall user experience. At the current stage, FWX has thoroughly audited the LBPs and partially audited the NFT membership aspect, ensuring the protocol's integrity and reliability.

By integrating these elements into a permissionless listing environment, FWX empowers global participation, promotes continuous innovation, and encourages an open financial landscape—free from traditional gatekeepers and accessible to anyone who wishes to engage with its diverse suite of decentralized financial tools.

Scope of Work

The security audit conducted does not replace the full security audit of the overall **FWX** protocol. The scope is limited to the **DeFi Permissionless** and their related smart contracts.

The security audit covered the components at this specific state:

Item	Description
Components	▪ *Core smart contracts* ▪ *Factory smart contracts* ▪ *NFT smart contracts* ▪ *Pool smart contracts* ▪ *Stakepool smart contracts* ▪ *Imported associated smart contracts and libraries*
Git Repository	▪ *https://github.com/forward-x/defi-permissionless-audit*
Audit Commit	▪ *90ca70341fb1cf977c7de0ce36e65864233d9f90 (branch: develop)*
Certified Commit	▪ *95fb8c80db8b6d239f1ff5b039ff076f3db1b3cb (branch: develop)*
Audited Files	▪ *contracts/src/*.sol*
Excluded Files/Contracts	▪ *contracts/src/stakepool/*.sol* ▪ *contracts/src/nft/*.sol* ▪ *contracts/src/libraries/*.sol* ▪ *contracts/src/helper/*.sol*

Remark: Our security best practices strongly recommend that the FWX team conduct a full security audit for both on-chain and off-chain components of its infrastructure and the interaction between them.

Auditors

Role	Staff List
Auditors	Anak Mirasing Kritsada Dechawattana Parichaya Thanawuthikrai Nattawat Songsom
Authors	Anak Mirasing Kritsada Dechawattana Parichaya Thanawuthikrai Nattawat Songsom
Reviewers	Sumedt Jitpukdebodin

Disclaimer

Our smart contract audit was conducted over a limited period and was performed on the smart contract at a single point in time. As such, the scope was limited to current known risks during the work period. The review does not indicate that the smart contract and blockchain software has no vulnerability exposure.

We reviewed the security of the smart contracts with our best effort, and we do not guarantee a hundred percent coverage of the underlying risk existing in the ecosystem. The audit was scoped only in the provided code repository. The on-chain code is not in the scope of auditing.

This audit report does not provide any warranty or guarantee, nor should it be considered an “approval” or “endorsement” of any particular project. This audit report should also not be used as investment advice nor provide any legal compliance.

Audit Result Summary

From the audit results and the remediation and response from the developer, Valix trusts that the **DeFi Permissionless** have sufficient security protections to be safe for use.

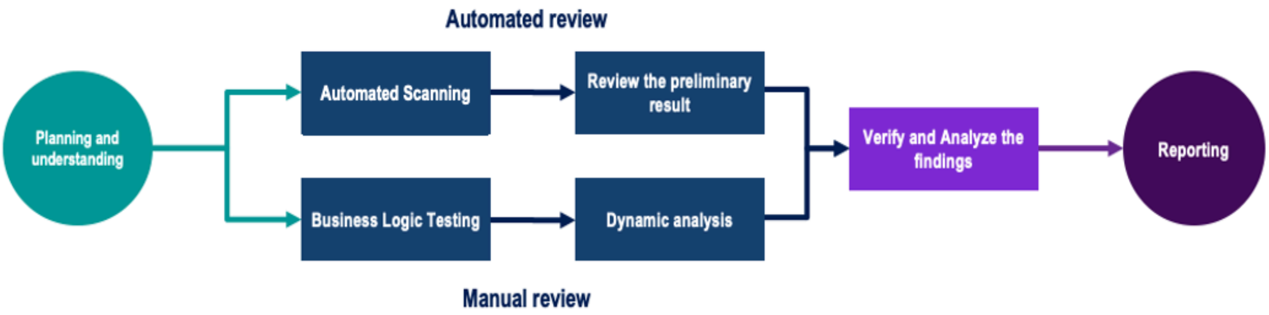
Initially, Valix was able to identify **54 issues** that were categorized from the "Critical" to "Informational" risk level in the given timeframe of the assessment. **Of these, the team was able to completely fix 42 issues and acknowledged 12 issues**. Below is the breakdown of the vulnerabilities found and their associated risk rating for each assessment conducted.

Target	Assessment Result					Reassessment Result				
	C	H	M	L	I	C	H	M	L	I
FWX DeFi Permissionless	11	7	18	3	15	0	4	5	0	3

Note: *Risk Rating* C *Critical,* H *High,* M *Medium,* L *Low,* I *Informational*

Methodology

The smart contract security audit methodology is based on Smart Contract Weakness Classification and Test Cases (SWC Registry), CWE, well-known best practices, and smart contract hacking case studies. Manual and automated review approaches can be mixed and matched, including business logic analysis in terms of the malicious doer's perspective. Using automated scanning tools to navigate or find offending software patterns in the codebase along with a purely manual or semi-automated approach, where the analyst primarily relies on one's knowledge, is performed to eliminate the false-positive results.



Planning and Understanding

- Determine the scope of testing and understanding of the application’s purposes and workflows.
- Identify key risk areas, including technical and business risks.
- Determine which sections to review within the resource constraints and review method – automated, manual or mixed.

Automated Review

- Adjust automated source code review tools to inspect the code for known unsafe coding patterns.
- Verify the tool's output to eliminate false-positive results, and adjust and re-run the code review tool if necessary.

Manual Review

- Analyzing the business logic flaws requires thinking in unconventional methods.
- Identify unsafe coding behavior via static code analysis.

Reporting

- Analyze the root cause of the flaws.
- Recommend improvements for secure source code.

Audit Items

We perform the audit according to the following categories and test names.

Category	ID	Test Name
Security Issue	**SEC01**	*Authorization Through tx.origin*
	SEC02	*Business Logic Flaw*
	SEC03	*Delegatecall to Untrusted Callee*
	SEC04	*DoS With Block Gas Limit*
	SEC05	*DoS with Failed Call*
	SEC06	*Function Default Visibility*
	SEC07	*Hash Collisions With Multiple Variable Length Arguments*
	SEC08	*Incorrect Constructor Name*
	SEC09	*Improper Access Control or Authorization*
	SEC10	*Improper Emergency Response Mechanism*
	SEC11	*Insufficient Validation of Address Length*
	SEC12	*Integer Overflow and Underflow*
	SEC13	*Outdated Compiler Version*
	SEC14	*Outdated Library Version*
	SEC15	*Private Data On-Chain*
	SEC16	*Reentrancy*
	SEC17	*Transaction Order Dependence*
	SEC18	*Unchecked Call Return Value*
	SEC19	*Unexpected Token Balance*
	SEC20	*Unprotected Assignment of Ownership*
	SEC21	*Unprotected SELFDESTRUCT Instruction*
	SEC22	*Unprotected Token Withdrawal*
	SEC23	*Unsafe Type Inference*
	SEC24	*Use of Deprecated Solidity Functions*
	SEC25	*Use of Untrusted Code or Libraries*
	SEC26	*Weak Sources of Randomness from Chain Attributes*
	SEC27	*Write to Arbitrary Storage Location*

Category	ID	Test Name
Functional Issue	**FNC01**	*Arithmetic Precision*
	FNC02	*Permanently Locked Fund*
	FNC03	*Redundant Fallback Function*
	FNC04	*Timestamp Dependence*
Operational Issue	**OPT01**	*Code With No Effects*
	OPT02	*Message Call with Hardcoded Gas Amount*
	OPT03	*The Implementation Contract Flow or Value and the Document is Mismatched*
	OPT04	*The Usage of Excessive Byte Array*
	OPT05	*Unenforced Timelock on An Upgradeable Proxy Contract*
Developmental Issue	**DEV01**	*Assert Violation*
	DEV02	*Other Compilation Warnings*
	DEV03	*Presence of Unused Variables*
	DEV04	*Shadowing State Variables*
	DEV05	*State Variable Default Visibility*
	DEV06	*Typographical Error*
	DEV07	*Uninitialized Storage Pointer*
	DEV08	*Violation of Solidity Coding Convention*
	DEV09	*Violation of Token (ERC20) Standard API*

Risk Rating

To prioritize the vulnerabilities, we have adopted the scheme of five distinct levels of risk: **Critical**, **High**, **Medium**, **Low**, and **Informational**, based on OWASP Risk Rating Methodology. The risk level definitions are presented in the table.

Risk Level	Definition
Critical	The code implementation does not match the specification, and it could disrupt the platform.
High	The code implementation does not match the specification, or it could result in losing funds for contract owners or users.
Medium	The code implementation does not match the specification under certain conditions, or it could affect the security standard by losing access control.
Low	The code implementation does not follow best practices or use suboptimal design patterns, which may lead to security vulnerabilities further down the line.
Informational	Findings in this category are informational and may be further improved by following best practices and guidelines.

The **risk value** of each issue was calculated from the product of the **impact** and **likelihood values**, as illustrated in a two-dimensional matrix below.

- **Likelihood** represents how likely a particular vulnerability is exposed and exploited in the wild.
- **Impact** measures the technical loss and business damage of a successful attack.
- **Risk** demonstrates the overall criticality of the risk.

Likelihood Impact	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Informational

The shading of the matrix visualizes the different risk levels. Based on the acceptance criteria, the risk levels "Critical" and "High" are unacceptable. Any issue obtaining the above levels must be resolved to lower the risk to an acceptable level.

Findings

System Trust Assumptions

Trust assumptions

The trust assumption in this context is that the **FWX DeFi permissionless** protocol allows the trusted operator to oversee the protocol.

It's important to note that, while the trusted operator is granted specific privileges to oversee the **FWX DeFi permissionless** protocol, special attention should be given to the account with the ***addressTimelockManager***, ***noTimelockManager*** and ***configTimelockManager*** role. These accounts have the authority to change the address of external calls, pause functionalities and change protocol configuration.

Furthermore, the trusted operator can execute actions without the need for a time-lock mechanism. This implies that any action within the scope of the trusted operator's authority will be carried out promptly.

The privileged roles

In the **FWX DeFi permissionless** protocol, privileged roles have special access to perform sensitive actions, relying on the trust placed in these roles to ensure the proper functioning and security of the system.

The **APHCore** contract:

- The **addressTimelockManager** account:
    - Can set the address of the *LogicStorage* contract.
- The **noTimelockManager** account:
    - Can pause and unpause several functionalities such as opening future positions etc.

The **CoreSetting** and **APHCoreSettingProxy** contract:

- The **addressTimelockManager** account:
    - Can set the address of the *Membership* contract.
    - Can set the address of the *PriceFeed* contract.
    - Can set the address of the *WETHHandler* contract.
    - Can set the address of the *FeeVault* contract.
    - Can approve several tokens to DEX routers.

  - Can set the whitelist for the collateral tokens, this determines whether the token can be used as collateral.
  - Can set several addresses as DEX routers.
- The **configTimelockManager** account:
  - Can set the address of the *FORWTradingVault* contract.
  - Can set the maximum leverage allowed for future positions.
  - Can set the percentage of interest to be splitted as *heldTokenInterest*.
  - Can set the percentage of *tradingFeeToLender*.
  - Can set the percentage of *auctionSpread*.
  - Can set the configuration of future positions such as minimum/maintenance margin, bounty fee to liquidator and protocol, minimum/maximum position size.
  - Can set the percentage of *liquidationFee*.
  - Can register new *APHPools*.
  - Can set the configuration of interaction with DEX routers such as max swap size, max price impact, max price different percent from the oracle.
  - Can set the configuration to get *FORW* token bonus such as *FORW* bonus amount, target position size to get the bonus.
  - Can set the swap fee rate of each DEX routers.
  - Can set the value of *forwStakingMultiplier* which will be used to determine whether the staking balance is enough to deposit more tokens.

The **FwxFactory** contract:

- The **addressTimelockManager** account:
  - Can set the address of the *FwxFactoryLogic* contract.
  - Can set the address of the *FwxFactorySetting* contract.
  - Can set the address of the *FwxFactoryValidator* contract.

The **FwxFactorySetting** contract:

- The **addressTimelockManager** account:
  - Can set the address of *ProxyAdmin* contract.

- Can set the address of the *LogicStorage* contract.
- Can set the address of *FORW* token contract.
- Can set the address of the *PriceFeed* contract.
- Can set the address of the *Membership* contract.

- The **configTimelockManager** account:
  - Can set the configuration related to validating new token pairs such as minimum DEX reserve required.
  - Can set the percentage of interest to be splitted as *heldTokenInterest*.
  - Can set the percentage of *tradingFeeToLender*.
  - Can set the percentage of *auctionSpread*.
  - Can set the maximum leverage allowed for future positions.
  - Can set the percentage of *liquidationFee*.
  - Can set the whitelist for the collateral tokens, this determines whether the token can be used as collateral.
  - Can set the configuration of future positions such as minimum/maintenance margin, bounty fee to liquidator and protocol, minimum/maximum position size.
  - Can set several addresses as DEX routers.
  - Can set the configuration of interaction with DEX routers such as max swap size, max price impact, max price different percent from the oracle.
  - Can set the swap fee rate of each DEX routers.
  - Can set the block time to be used in time based calculations.
  - Can set the treasury account address.
  - Can set the borrowing rates and the utilization rates percentage to calculate annual percentage rates (APRs) and the borrowing interest.

The **FwxFactorySettingProxy** contract:

- The **configTimelockManager** account:
  - Can set the treasury account address.

The **APHPool** contract:

- The **noTimelockManager** account:
  - Can pause and unpause several functionalities such as withdrawing tokens from the lending pool etc.

The **PoolLending** contract:

- The **noTimelockManager** account:
  - Can be used by the *FWXFactory* contract for the initial token deposit to the pool.

The **PoolSetting** contract:

- The **addressTimelockManager** account:
  - Can set the address of the *LogicStorage* contract.
  - Can set the address of the *WETHHandler* contract.
  - Can set the address of the *Membership* contract.
- The **configTimelockManager** account:
  - Can set the borrowing rates and the utilization rates percentage to calculate annual percentage rates (APRs) and the borrowing interest.

The **Vault** contract:

- The **addressTimelockManager** account:
  - Can approve several tokens to the target pool contract.
  - Can approve several tokens to the target core contract.

The **FeeVault** contract:

- The **addressTimelockManager** account:
  - Can set the address of the profit receiver account, this account has the access to withdraw profit from this vault.
  - Can set the address of the auction fee receiver account, this account has the access to withdraw auction fee from this vault.
  - Can set the address of the *FWXFactory* contract.

The **InterestVault** contract:

- The **addressTimelockManager** account:

    - Can set the address of the interest token.
    - Can set the treasury account address.
    - Can set the address of the *APHCore* contract.
    - Can approve several tokens to the target pool contract.
- The **noTimelockManager** account:
    - Can trigger the withdrawal process, moves token balance in *actualTokenInterestProfit* state to the treasury address.

The **PriceFeed** and **PriceFeedL2** contract:

- The **configTimelockManager** account:
    - Can set the address of the external *PriceFeed* Oracle contract and decimal value for several tokens.
    - Can set the acceptable stale period for several tokens.
- The **noTimelockManager** account:
    - Can pause and unpause the query USD price rate functionality.

The **LogicStorage** contract:

- The **addressTimelockManager** account:
    - Can set the address of the *APHCoreProxy* contract.
    - Can set the address of the *APHCore* contract.
    - Can set the address of the *CoreSetting* contract.
    - Can set the address of the *CoreFutureWallet* contract.
    - Can set the address of the *CoreFutureOpening* contract.
    - Can set the address of the *CoreFutureClosing* contract.
    - Can set the address of the *CoreSwapping* contract.
    - Can set the address of the *APHPool* contract.
    - Can set the address of the *PoolLending* contract.
    - Can set the address of the *PoolBorrowing* contract.

Review Findings Summary

The table below shows the summary of our assessments.

No.	Issue	Risk	Status	Functionality is in use
1	Uninitialized Implementation Contracts	Critical	Fixed	In use
2	Invalid Target Supply	Critical	Fixed	In use
3	Unrestricted Access To The setWeth And setWethHandler Functions	Critical	Fixed	In use
4	Incorrect Token Validation In isUnderlyingValid Check	Critical	Fixed	In use
5	Inability To Withdraw Token Actual Profit	Critical	Fixed	In use
6	Inability To Transfer Manager Roles	Critical	Fixed	In use
7	Inability To Modify Crucial Contract States	Critical	Fixed	In use
8	Potential Denial Of Service On APHPool	Critical	Fixed	In use
9	Loss Tracking Precision Mismatch In APHCore	Critical	Fixed	In use
10	Incorrect Margin Position Validation For Collateral Withdrawal	Critical	Fixed	In use
11	Lock Of The Borrow Token In The APHCore Due To Double Subtracted Fee	Critical	Fixed	In use
12	Loss Of Claimable Interest In Rounding Down Issue	High	Acknowledged	In use
13	Potential Of Global Setting Precisions Mismatch With Token Precisions	High	Acknowledged	In use
14	Potential Inability To Withdraw Principal Token Due To Arithmetic Underflow Revert	High	Fixed	In use
15	Potential Rounding Down For SwapFee Calculation	High	Fixed	In use
16	Potential Locking Of bountyFeeToLiquidator Within The APHPool	High	Fixed	In use
17	Inaccessibility Of Markets Due To Unsupported Tokens Tn Price Feed	Medium	Acknowledged	In use
18	The Chainlink Oracle Rate Has The Potential To Be Either Negative Or Zero	Medium	Fixed	In use
19	Not Support Chainlink L2 Sequencer Down	Medium	Fixed	In use
20	Compatibility Issue With USDT Allowance	Medium	Fixed	In use

	Mechanism In Vault			
21	Missing Validating address(0) In Low-Level Delegatecall	Medium	Fixed	In use
22	Potential Inconsistency Of Crucial States	Medium	Fixed	In use
23	Over Deposited Amounts Are Non-Refundable	Medium	Fixed	In use
24	Risk of Withdrawal Restrictions	Medium	Acknowledged	In use
25	Possibly Inconsistent Setting With The Actual Swap Fee	Medium	Acknowledged	In use
26	Potentially Underflow Revert On Bounty Fee Distribution	Medium	Fixed	In use
27	Potentially Underflow Revert On Profit Distribution	Medium	Fixed	In use
28	Potentially Underflow Revert On The withdrawTokenInterest Function	Medium	Fixed	In use
29	Lack Of Support For Multiple Routers Configuration	Medium	Acknowledged	In use
30	Inconsistency In Fee, Trading Fee And Auction Spread Validation	Medium	Fixed	In use
31	Improperly Getting Total Token Interest	Medium	Fixed	In use
32	Incorrect Behavior Of Usable NFT	Medium	Fixed	In use
33	Incorrect collateralSwappedAmount Return Event Emission Value	Medium	Fixed	In use
34	Recommended Following Best Practices For Upgradeable Smart Contracts	Low	Fixed	In use
35	Unsafe ABI Encoding	Low	Fixed	In use
36	Incomplete Legacy Data Removal In Utils Rates	Low	Fixed	In use
37	Recommended Removing Unused Code	Informational	Fixed	In use
38	Misspelled Variable And Parameter Names	Informational	Fixed	In use
39	Recommended Improving Comments To Reflect The Code	Informational	Fixed	In use
40	Incorrectly Emitted Event Value	Informational	Fixed	In use
41	Enhancing Library Compatibility with Non-upgradeable Contracts	Informational	Fixed	In use
42	Recommended Improving The Error Messages	Informational	Fixed	In use
43	Incorrect Filename	Informational	Fixed	In use

44	Unnecessary Data Overriding with _delegateCall	Informational	Fixed	In use
45	Inconsistency In Burnable Amount Logic	Informational	Fixed	In use
46	Deposit Native Token Failure Due To Requirement Conflict	Informational	Fixed	In use
47	Inability To Disable The Routers After Being Set	Informational	Fixed	In use
48	Mismatched NFT Owner Event Emission	Informational	Fixed	In use
49	Price Impact Due To Low Liquidity: DEX vs Oracle Price Discrepancy	Informational	Acknowledged	In use
50	Recommended Enforcing Checks-Effects-Interactions Pattern	Informational	Acknowledged	In use

The table below shows the issues from the reassessment process.

No.	Issue	Risk	Status	Functionality is in use
1	Lack Of Price Slippage Control Mechanism	High	Acknowledged	In use
2	Lack Of Lender Loss Tracking	High	Acknowledged	In use
3	Potential Over-Distribution Of Lending Bonuses	Medium	Acknowledged	In use
4	Out Of Audit Scope	Informational	Acknowledged	In use

The statuses of the issues are defined as follows:

Fixed: The issue has been completely resolved and has no further complications.

Partially Fixed: The issue has been partially resolved.

Acknowledged: The issue's risk has been reported and acknowledged.

Detailed Result

This section provides all issues that we found in detail.

No. 1	Uninitialized Implementation Contracts		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/factory/FwxFactory.sol* *contracts/src/factory/proxy/FwxFactorySettingProxy.sol*		
Locations	*FwxFactory.initialize L: 23 - 56* *FwxFactory.setFwxFactorySetting L: 64 - 68* *FwxFactorySettingProxy.setProxyAdmin L: 10 - 16*		

Detailed Issue

The *FwxFactory* contract is designed to be implementation contracts supporting an upgradeable feature. That is, these implementation contracts will be the logic contracts for their proxy contracts.

We found that the *FwxFactory* implementation contract would be left uninitialized when it is deployed resulting in being taken over by an attacker. As a result, the attacker can perform a denial-of-service attack rendering the proxy contracts unusable.

To understand this issue, consider the following attack scenario of the *FwxFactory* implementation contract.

1. The *FwxFactory* implementation and proxy contracts are deployed and set up by a developer.
2. An attacker discovers the *FwxFactory* implementation contract uninitialized. He takes over the implementation contract by calling the initialize function (L23-56 in *FwxFactory*). As a result, the *addressTimelockManager* state variable is set to the attacker address (L35 in *FwxFactory*).
3. The attacker deploys a *Rogue* contract implementing a *(mock) setProxyAdmin* function.
4. The attacker makes a call to the *FwxFactory's setFwxFactorySetting* function to set the *fwxFactorySetting* state variable to the previously deployed Rogue contract address (L64-68 in *FwxFactory*).
5. The attacker executes the *FwxFactory's setProxyAdmin* function which would make a delegatecall to the (mock) *setProxyAdmin* function of the Rogue contract (L10-16 in *FwxFactorySettingProxy*).
6. The (mock) *setProxyAdmin* function invokes the *selfdestruct* instruction resulting in removing the contract code from the *FwxFactory* implementation contract address.
7. The *FwxFactory* proxy contract becomes unusable since its implementation contract was destroyed.

We consider this issue critical since suddenly after the *FwxFactory* implementation contracts are destroyed, its proxy contract would no longer operate.

FwxFactory.sol

```
function initialize(
   address _logicStorage,
   address _fwx,
   address _priceFeed,
   address _membership,
   address _weth,
   address _wethHandler,
   address _feeVault,
   uint256 _blockTime
) external initializer {
   noTimelockManager = msg.sender;
   configTimelockManager = msg.sender;
   addressTimelockManager = msg.sender;

   logicStorage = _logicStorage;
   fwx = _fwx;
   priceFeed = _priceFeed;
   membership = _membership;
   weth = _weth;
   wethHandler = _wethHandler;
   blockTime = _blockTime;
   feeVault = _feeVault;

   emit TransferNoTimelockManager(address(0), msg.sender);
   emit TransferConfigTimelockManager(address(0), msg.sender);
   emit TransferAddressTimelockManager(address(0), msg.sender);

   emit SetLogicStorage(msg.sender, address(0), _logicStorage);
   emit SetFwxAddress(msg.sender, address(0), _fwx);
   emit SetPriceFeedAddress(msg.sender, address(0), _priceFeed);
   emit SetMembershipAddress(msg.sender, address(0), _membership);
   emit SetWethAddress(msg.sender, address(0), _weth);
   emit SetWethHandlerAddress(msg.sender, address(0), _wethHandler);
}
```

Listing 1.1 The *initialize* function in the *FwxFactory*

FwxFactory.sol

```
function setFwxFactory(address _fwxFactory) external onlyAddressTimelockManager
{
    address oldFwxFactort = fwxFactory;
    fwxFactory = _fwxFactory;
    emit SetFwxFactory(msg.sender, oldFwxFactort, fwxFactory);
}
```

Listing 1.2 The *setFwxFactorySetting* function in *FwxFactory*

FwxFactorySettingProxy.sol

```
function setProxyAdmin(address _proxyAdmin) external override {
    bytes memory data = abi.encodeWithSelector(
        IFwxFactorySetting.setProxyAdmin.selector,
        _proxyAdmin
    );
    data = _delegatecall(fwxFactorySetting, data);
}
```

Listing 1.3 The *setProxyAdmin* function in *FwxFactorySettingProxy*

Recommendations

To address this issue, we recommend adding the *constructor* like the code snippet below to the *FwxFactory* implementation contract.

The added *constructor* guarantees that the implementation contract would be automatically initialized during its deployment, closing the room for an attacker to take over the implementation contract.

FwxFactory.sol

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Listing 1.4 Adding *constructor* with *_disableInitializers()* to the *FwxFactory*

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

FwxFactory.sol

```
constructor() {
    _disableInitializers();
}
```

Listing 1.5 The improved *constructor* of the *FwxFactory* contract

No. 2	Invalid Target Supply		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/factory/logic/FwxFactoryLogic.sol* *contracts/src/pool/logic/PoolBaseFunc.sol* *contracts/src/pool/logic/PoolSetting.sol*		
Locations	*PoolBaseFunc._getNextBorrowingInterest L: 55 - 92* *FwxFactoryLogic._setupConfig L: 166 - 221* *PoolSetting.setBorrowInterestParams L: 9 - 32*		

Detailed Issue

The *_getNextBorrowingInterest* function uses the *targetSupply* variable (L69 in the code snippet below) of the *PoolBase* contract to calculate the *w* variable which is used to calculate the borrowing interest later.

PoolBaseFunc.sol

```
55  function _getNextBorrowingInterest(
56      uint256 newBorrowAmount
57  ) internal view returns (uint256 nextInterestRate) {
58      uint256[10] memory localUtils = utils;
59      uint256[10] memory localRates = rates;
60
61      nextInterestRate = localRates[0];
62
63      if (pTokenTotalSupply == 0) {
64          return nextInterestRate;
65      }
66
67      uint256 w = (MathUpgradeable.max(
68          PRECISION_UNIT,
69          (targetSupply * PRECISION_UNIT) / pTokenTotalSupply
70      ) * WEI_UNIT) / PRECISION_UNIT;
71
72     // (...SNIPPED...)
```

Listing 2.1 The *_getNextBorrowingInterest* function of the *PoolBaseFunc* contract

We noticed that the *targetSupply* variable is set once within the *_setupConfigs* function of the *FwxFactoryLogic* contract with the *targetSupply* variable of the *FwxFactoryLogic* contract when creating a market. However, **we found that the *targetSupply* variable (L181 and 182 in code snippet 2.2) of the *FwxFactoryLogic* contract is always zero because there is no setter function**.

Consequently, the *_getNextBorrowingInterest* function always uses the *PRECISION_UNIT* (L68 in code snippet 2.1) to calculate the borrowing interest.

FwxFactoryLogic.sol

```
166  function _setupConfigs(
167      address core,
168      address collateralPool,
169      address underlyingPool,
170      address collateralToken,
171      address underlyingToken
172  ) internal {
173      IAPHCoreSetting coreSetting = IAPHCoreSetting(core);
174
175      /* ---------------------------------- pool
     ---------------------------------- */
176      coreSetting.registerNewPool(collateralPool);
177      coreSetting.registerNewPool(underlyingPool);
178
179      IAPHPoolSetting collateralPoolSetting = IAPHPoolSetting(collateralPool);
180      IAPHPoolSetting underlyingPoolSetting = IAPHPoolSetting(underlyingPool);
181      collateralPoolSetting.setBorrowInterestParams(rates, utils, targetSupply);
182      underlyingPoolSetting.setBorrowInterestParams(rates, utils, targetSupply);

         // (...SNIPPED...)
```

Listing 2.2 The *_setupConfig* function of the *FwxFactoryLogic* contract

PoolSetting.sol

```
 9   function setBorrowInterestParams(
10       uint256[] memory _rates,
11       uint256[] memory _utils,
12       uint256 _targetSupply
13   ) external onlyConfigTimelockManager {
14       require(_rates.length == _utils.length, "PoolSetting/length-not-equal");
15       require(_rates.length <= 10, "PoolSetting/length-too-high");
16       require(_utils[0] == 0, "PoolSetting/invalid-first-util");
17       require(_utils[_utils.length - 1] == WEI_PERCENT_UNIT,
     "PoolSetting/invalid-last-util");
18
19       for (uint256 i = 1; i < _rates.length; i++) {
20           require(_rates[i - 1] <= _rates[i], "PoolSetting/invalid-rate");
21           require(_utils[i - 1] < _utils[i], "PoolSetting/invalid-util");
22       }
23
24       for (uint256 i = 0; i < _rates.length; i++) {
        rates[i] = _rates[i];
        utils[i] = _utils[i];
    }
    targetSupply = _targetSupply;
    utilsLen = _utils.length;

    emit SetBorrowInterestParams(msg.sender, _rates, _utils, targetSupply);
}
```

Listing 2.3 The *setBorrowInterestParams* function of the *PoolSetting* contract

Recommendations

We recommend invoking the setBorrowInterestParams function with the given targetSupply precision in the same as the target token precision.

Reassessment

The *FWX* team removes the ***targetSupply*** from *FwxFactoryBase* contract and invokes the *setBorrowInterestParams* function with a constant 0 instead.

FwxFactoryLogic.sol

```
function _setupConfigs(
    address core,
    address collateralPool,
    address underlyingPool,
    address collateralToken,
    address underlyingToken
) internal {
    IAPHCoreSetting coreSetting = IAPHCoreSetting(core);

    /* ---------------------------------- pool
---------------------------------- */
    coreSetting.registerNewPool(collateralPool);
    coreSetting.registerNewPool(underlyingPool);

    IAPHPoolSetting collateralPoolSetting = IAPHPoolSetting(collateralPool);
    IAPHPoolSetting underlyingPoolSetting = IAPHPoolSetting(underlyingPool);
    collateralPoolSetting.setBorrowInterestParams(rates, utils, 0);
    underlyingPoolSetting.setBorrowInterestParams(rates, utils, 0);

    // (...SNIPPED...)
```

Listing 2.4 The *_setupConfigs* function of the *FwxFactoryLogic* contract

No. 3	Unrestricted Access To The setWeth And setWethHandler Functions		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/factory/logic/FwxFactorySetting.sol*		
Locations	*FwxFactorySetting.setWeth L: 51 - 53* *FwxFactorySetting.setWethHandlerAddress L: 62 - 64*		

Detailed Issue

We found that the ***setWeth*** and ***setWethHandlerAddress*** functions of the ***FwxFactorySetting*** contract do not have access restrictions, allowing anyone to invoke and set the *weth* and *wethHandler* addresses.

FwxFactorySetting.sol

```
51  function setWeth(address _weth) external {
52          _setWeth(_weth);
53  }

    function _setWeth(address _weth) internal {
            address oldWETH = weth;
            weth = _weth;

            emit SetWethAddress(msg.sender, oldWETH, weth);
    }

62  function setWethHandlerAddress(address _wethHandler) external {
63          _setWethHandlerAddress(_wethHandler);
64  }

    function _setWethHandlerAddress(address _wethHandler) internal {
            address oldWethHandler = wethHandler;
            wethHandler = _wethHandler;

            emit SetWethHandlerAddress(msg.sender, oldWethHandler, membership);
    }
```

Listing 3.1 The *setWeth and setWethHandlerAddress* functions of the *FwxFactorySetting* contract

Recommendations

We recommend applying the restriction to the ***setWeth*** and ***setWethHandlerAddress*** functions of the ***FwxFactorySetting*** contract.

Reassessment

The *FWX* team adopted our recommended code to fix this issue by applying the *onlyAddressTimelockManager* modifier to the ***setWeth*** and ***setWethHandlerAddress*** functions.

FwxFactorySetting.sol

```
function setWeth(address _weth) external onlyAddressTimelockManager {
    address oldWETH = weth;
    weth = _weth;

    emit SetWethAddress(msg.sender, oldWETH, weth);
}

function setWethHandlerAddress(address _wethHandler) external onlyAddressTimelockManager {
    address oldWethHandler = wethHandler;
    wethHandler = _wethHandler;

    emit SetWethHandlerAddress(msg.sender, oldWethHandler, wethHandler);
}
```

Listing 3.2 The improved *setWeth* and *setWethHandlerAddress* functions of the *FwxFactorySetting* contract

No. 4	Incorrect Token Validation In isUnderlyingValid Check		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/factory/logic/FwxFactoryValidator.sol*		
Locations	*FwxFactoryValidator._validateDex L: 169 - 172*		

Detailed Issue

The *_validateDex* function is designed to validate the collateral token and the underlying token when creating a new market. The validation process includes checking the reserves of both tokens on the Dex.

We found that the *_validateDex* function mistakenly validates the collateral token instead of the underlying token in the underlying token validation process (L171-172 in code snippet below). As a result, the underlying token is not properly validated when creating a market.

FwxFactoryValidator.sol

```
147 function _validateDex(
148     address _collateralToken,
149     address _underlyingToken
150 ) internal view returns (bool isPairExist, bool isCollateralValid, bool
    isUnderlyingValid) {
       // (...SNIPPED...)
169    isCollateralValid =
           reserve0 >= _parseTokenPrecisions(_collateralToken,
    cfg.minCollateralTokenDEXReserve);
170    isUnderlyingValid =
           reserve1 >= _parseTokenPrecisions(_collateralToken,
    cfg.minUnderlyingTokenDEXReserve);
```

Listing 4.1 The *_validateDex* function in *FwxFactoryValidator*

Recommendations

To address this issue, **we recommend using the *_underlyingToken* variable for the underlying token validation process**.

FwxFactoryValidator.sol

```
function _validateDex(
    address _collateralToken,
    address _underlyingToken
) internal view returns (bool isPairExist, bool isCollateralValid, bool
isUnderlyingValid) {
   // (...SNIPPED...)
   isCollateralValid =
       reserve0 >= _parseTokenPrecisions(_collateralToken,
cfg.minCollateralTokenDEXReserve);
   isUnderlyingValid =
       reserve1 >= _parseTokenPrecisions(_underlyingToken,
cfg.minUnderlyingTokenDEXReserve);
```

Listing 4.2 Validating underlying token with *minUnderlyingTokenDEXReserve*

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team uses the *_underlyingToken* variable for the underlying token validation process to fix this issue as shown in the code snippet below.

FWXFactoryValidator.sol

```
function _validateDex(
        address _collateralToken,
        address _underlyingToken
    ) internal view returns (bool isPairExist, bool isCollateralValid, bool
isUnderlyingValid) {

// (...SNIPPED...)

isCollateralValid =
        reserve0 >= _parseTokenPrecisions(_collateralToken,
cfg.minCollateralTokenDEXReserve);
isUnderlyingValid =
        reserve1 >= _parseTokenPrecisions(_underlyingToken,
cfg.minUnderlyingTokenDEXReserve);
```

Listing 4.3 The fixed *_validateDex* function of the *FWXFactoryValidator* contract

No. 5	Inability To Withdraw Token Actual Profit		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/InterestVault.sol*		
Locations	*InterestVault.withdrawActualProfit L: 112 - 114*		

Detailed Issue

The *APHCore*, *APHPool*, and *InterestVault* (deployed along with the *APHPool*) contracts created by the *FwxFactory* contract are unable to configure their crucial setting that restriction by the following *modifiers* after the contract's creation:

- ***onlyNoTimelockManager***
- ***configTimelockManager***
- ***addressTimelockManager***

Since at the creation stage of the permissionless *APHCore*, *APHPool,* and *InterestVault* contracts, the *FwxFactory* will be the deployer of these contracts and it does not have the logic to configure the setting of each market after their deployment.

As a result, we found the **inability to withdraw the token actual profit from the *InterestVault* contracts of each *APHPool*** as follows:

- The *treasuryAddress* address is not set at the contract creation config and cannot be set after that as the restriction of ***onlyAddressTimelockManager***
- unable to call the ***withdrawActualProfit*** function by restriction of ***onlyNoTimelockManager***

InterestVault.sol

```
contract InterestVault is InterestVaultEvent, Ownable, SelectorPausable, ManagerTimelock {

    // (...SNIPPED...)
    function setTokenAddress(address _address) external onlyAddressTimelockManager {
        address oldAddress = tokenAddress;
        tokenAddress = _address;

```

```solidity
        emit SetTokenAddress(msg.sender, oldAddress, tokenAddress);
    }

    function setTreasuryAddress(address _address) external onlyAddressTimelockManager {
        address oldAddress = treasuryAddress;
        treasuryAddress = _address;

        emit SetTreasuryAddress(msg.sender, oldAddress, treasuryAddress);
    }

    function setProtocolAddress(address _address) external onlyAddressTimelockManager {
        address oldAddress = protocolAddress;
        protocolAddress = _address;

        emit SetProtocolAddress(msg.sender, oldAddress, protocolAddress);
    }

    // (...SNIPPED...)

    function withdrawActualProfit() external onlyNoTimelockManager returns (uint256) {
        return _withdrawActualProfit();
    }

    // (...SNIPPED...)

    function _withdrawActualProfit() internal returns (uint256) {
        uint256 tmpInterestProfit = actualTokenInterestProfit;
        actualTokenInterestProfit = 0;

        IERC20(tokenAddress).safeTransfer(treasuryAddress, tmpInterestProfit);
        emit WithdrawActualProfit(msg.sender, treasuryAddress, tmpInterestProfit);
        return tmpInterestProfit;
    }
}
```

Listing 5.1 The *withdrawActualProfit* function of the *InterestVault* contract

Recommendations

We recommend the team revising the access controls for the *withdrawActualProfit* function and the mechanism for setting the *treasuryAddress* state variable to ensure enhanced operational flexibility and security.

Reassessment

The *FWX* team introduced the new *_transferManagers* function (L236 - 263 in code snippet 5.2) that enables to transfer admins to another account. Then call this function within the create market process (L59 in code snippet 5.2) to transfer admins from the factory contract to the actual admin account instead.

FwxFactoryLogic.sol

```
 32  function createMarket(
 33      uint256 nftId,
 34      address collateralToken,
 35      address underlyingToken,
 36      uint256 collateralTokenSent,
 37      uint256 underlyingTokenSent
 38  ) external returns (address core, address collateralPool, address
     underlyingPool) {
 39      if (collateralToken == weth || underlyingToken == weth)
 40          revert FwxFactory_TokenNotAllowed(weth);
 41
 42      _validateMarketCreation(
 43          msg.sender,
 44          collateralToken,
 45          underlyingToken,
 46          collateralTokenSent,
 47          underlyingTokenSent
 48      );
 49      (core, collateralPool, underlyingPool) = _createMarket(collateralToken,
     underlyingToken);
 50
 51      //// setup APHCore and APHPool
 52      _setupConfigs(core, collateralPool, underlyingPool, collateralToken,
     underlyingToken);
 53
 54      //// Add liquidity
 55      _addLiquidity(nftId, collateralPool, collateralToken, collateralTokenSent);
 56      _addLiquidity(nftId, underlyingPool, underlyingToken, underlyingTokenSent);
 57
 58      //// transfer roles from FwxFactory to managers
 59      _transferManagers(core, collateralPool, underlyingPool);
 60  }

     // (...SNIPPED...)

236  function _transferManagers(
237      address core,
238      address collateralPool,
239      address underlyingPool
240  ) private {
241      ManagerTimelock manager = ManagerTimelock(core);
242      manager.transferNoTimelockManager(noTimelockManager);
```

```
    manager.transferConfigTimelockManager(configTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(collateralPool);
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferConfigTimelockManager(configTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(underlyingPool);
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferConfigTimelockManager(configTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(IAPHPool(collateralPool).interestVaultAddress());
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(IAPHPool(underlyingPool).interestVaultAddress());
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);
}
```

Listing 5.2 The transferring admins process of *FwxFactoryLogic* contract

No. 6	Inability To Transfer Manager Roles		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/InterestVault.sol* *contracts/src/pool/APHPool.sol* *contracts/src/core/APHCore.sol* *contracts/src/etc/ManagerTimelock.sol* *contracts/src/etc/ManagerTimelockUpgradeable.sol*		
Locations	*The transferNoTimelockManager function of associated files* *The transferConfigTimelockManager function of associated files* *The transferAddressTimelockManagerfunction of associated files*		

Detailed Issue

In the initial setup of the permissionless *APHCore*, *APHPool*, and *InterestVault* contracts deployed by *FwxFactory*, critical management roles such as *noTimelockManager*, *configTimelockManager*, and *addressTimelockManager* are established (e.g., L36 - 38 in code snippet 6.1).

However, **we found that the *FwxFactory* lacks the capability to execute following functions required for transferring the management roles**

1. The *transferNoTimelockManager* function
2. The *transferConfigTimelockManager* function
3. The *transferAddressTimelockManager* function

This prevents the transition of management roles to new entities, potentially limiting the flexibility of administrative actions and impacting the overall governance of the contracts.

APHPool.sol

```
20 function initialize(
21     address _logicStorage,
22     address _tokenAddress,
23     address _coreAddress,
24     address _membershipAddress,
25     address _wethAddress,
26     address _wethHandlerAddress,
27     uint256 _blockTime
28 ) external virtual initializer {
    require(_tokenAddress != address(0), "APHPool/initialize/tokenAddress-zero-address");
    require(_coreAddress != address(0), "APHPool/initialize/coreAddress-zero-address");
    require(_membershipAddress != address(0), "APHPool/initialize/membership-zero-address");
    tokenAddress = _tokenAddress;
    coreAddress = _coreAddress;
    membershipAddress = _membershipAddress;
    logicStorageAddress = _logicStorage;
    noTimelockManager = msg.sender;
    configTimelockManager = msg.sender;
    addressTimelockManager = msg.sender;

    interestVaultAddress = address(new InterestVault(tokenAddress, coreAddress, msg.sender));
    require(_blockTime != 0, "_blockTime cannot be zero");
       BLOCK_TIME = _blockTime;

    // (...SNIPPED...)
}
```

Listing 6.1 The *initialize* function of the *APHPool* contract

ManagerTimelockUpgradeable.sol

```
contract ManagerTimelockUpgradeable {

   // (...SNIPPED...)

   function transferNoTimelockManager(address _address) public virtual onlyNoTimelockManager {
        require(_address != address(0), "Manager/new-manager-is-the-zero-address");
        _transferNoTimelockManager(_address);
   }

   function transferConfigTimelockManager(
       address _address
   ) public virtual onlyConfigTimelockManager {
       require(_address != address(0), "Manager/new-manager-is-the-zero-address");
       _transferConfigTimelockManager(_address);
   }

   function transferAddressTimelockManager(
       address _address
   ) public virtual onlyAddressTimelockManager {
       require(_address != address(0),
"Manager/new-manager-is-the-zero-address");
            _transferAddressTimelockManager(_address);
    }

    // (...SNIPPED...)
}
```

Listing 6.2 The *functions responsible for transferring manager roles* of the *ManagerTimelockUpgradeable* contract

Recommendations

We recommend that the team either enables *FwxFactory* to invoke functions for transferring management roles or revises the access control mechanism governing these transfers to align with the protocol's business requirements. This adjustment will ensure that management roles can be effectively transitioned as needed to support the protocol's governance and flexibility.

Reassessment

The *FWX* team introduced the new *_transferManagers* function (L236 - 263) that enables to transfer admins to another account. Then call this function within the create market process (L59) to transfer admins from the factory contract to the actual admin account instead.

FwxFactoryLogic.sol

```
32  function createMarket(
33      uint256 nftId,
34      address collateralToken,
35      address underlyingToken,
36      uint256 collateralTokenSent,
37      uint256 underlyingTokenSent
38  ) external returns (address core, address collateralPool, address
    underlyingPool) {
39      if (collateralToken == weth || underlyingToken == weth)
40          revert FwxFactory_TokenNotAllowed(weth);
41
42      _validateMarketCreation(
43          msg.sender,
44          collateralToken,
45          underlyingToken,
46          collateralTokenSent,
47          underlyingTokenSent
48      );
49      (core, collateralPool, underlyingPool) = _createMarket(collateralToken,
    underlyingToken);
50
```

```
    //// setup APHCore and APHPool
    _setupConfigs(core, collateralPool, underlyingPool, collateralToken, underlyingToken);

    //// Add liquidity
    _addLiquidity(nftId, collateralPool, collateralToken, collateralTokenSent);
    _addLiquidity(nftId, underlyingPool, underlyingToken, underlyingTokenSent);

    //// transfer roles from FwxFactory to managers
    _transferManagers(core, collateralPool, underlyingPool);
}

// (...SNIPPED...)

function _transferManagers(
    address core,
    address collateralPool,
    address underlyingPool
) private {
    ManagerTimelock manager = ManagerTimelock(core);
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferConfigTimelockManager(configTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(collateralPool);
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferConfigTimelockManager(configTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(underlyingPool);
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferConfigTimelockManager(configTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(IAPHPool(collateralPool).interestVaultAddress());
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(IAPHPool(underlyingPool).interestVaultAddress());
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);
}
```

Listing 6.3 The transferring admins process of *FwxFactoryLogic* contract

No. 7	Inability To Modify Crucial Contract States		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/logic/PoolSetting.sol* *contracts/src/core/APHCore.sol* *contracts/src/core/logic/CoreSetting.sol*		
Locations	*Several functions throughout multiple contracts*		

Detailed Issue

Similar to ***No.5 and No.6*** issues, we encountered a **limitation in the permissionless *APHCore*, *APHPool*, and *InterestVault* contracts, which are deployed by *FwxFactory*.**

Due to the way these contracts are initialized, there's a restriction in modifying crucial state variables. This restriction arises from the management role modifiers applied during the contract deployment phase.

Consequently, this issue restricts the flexibility to adjust crucial contract states post-deployment in *APHCore*, *APHPool,* and *InterestVault*, potentially impacting the protocol's adaptability and governance.

PoolSetting.sol

```
contract PoolSetting is PoolBaseFunc, PoolSettingEvent {
   function setBorrowInterestParams(
       uint256[] memory _rates,
       uint256[] memory _utils,
       uint256 _targetSupply
   ) external onlyConfigTimelockManager {
      // (...SNIPPED...)
   }

   function setLogicStorageAddress(address _address) external onlyAddressTimelockManager {
       // (...SNIPPED...)
   }

   function setWETHHandler(address _address) external onlyAddressTimelockManager {
       // (...SNIPPED...)
   }

   function setMembershipAddress(address _address) external
onlyAddressTimelockManager {
       // (...SNIPPED...)
   }
}
```

Listing 7.1 The *crucial setter* functions of the *PoolSetting* contract

Recommendations

We advise the team to review and potentially revise the access control mechanisms related to the modification of crucial state variables in *APHCore*, *APHPool,* and *InterestVault*. One approach could be to provide *FwxFactory* or another designated entity with the capabilities to adjust these states or to refine the role-based access control to offer more flexibility in governance and protocol management.

Reassessment

The *FWX* team introduced the new *_transferManagers* function (L236 - 263 in code snippet 7.2) that enables to transfer admins to another account. Then call this function within the create market process (L59 in code snippet 7.2) to transfer admins from the factory contract to the actual admin account instead.

FwxFactoryLogic.sol

```
32 function createMarket(
33     uint256 nftId,
34     address collateralToken,
35     address underlyingToken,
36     uint256 collateralTokenSent,
37     uint256 underlyingTokenSent
38 ) external returns (address core, address collateralPool, address
   underlyingPool) {
39     if (collateralToken == weth || underlyingToken == weth)
40         revert FwxFactory_TokenNotAllowed(weth);
41 
42     _validateMarketCreation(
43         msg.sender,
44         collateralToken,
45         underlyingToken,
46         collateralTokenSent,
47         underlyingTokenSent
48     );
49     (core, collateralPool, underlyingPool) = _createMarket(collateralToken,
   underlyingToken);
50 
51     //// setup APHCore and APHPool
```

```
    _setupConfigs(core, collateralPool, underlyingPool, collateralToken, underlyingToken);

    //// Add liquidity
    _addLiquidity(nftId, collateralPool, collateralToken, collateralTokenSent);
    _addLiquidity(nftId, underlyingPool, underlyingToken, underlyingTokenSent);

    //// transfer roles from FwxFactory to managers
    _transferManagers(core, collateralPool, underlyingPool);
}

// (...SNIPPED...)

function _transferManagers(
    address core,
    address collateralPool,
    address underlyingPool
) private {
    ManagerTimelock manager = ManagerTimelock(core);
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferConfigTimelockManager(configTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(collateralPool);
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferConfigTimelockManager(configTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(underlyingPool);
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferConfigTimelockManager(configTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(IAPHPool(collateralPool).interestVaultAddress());
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);

    manager = ManagerTimelock(IAPHPool(underlyingPool).interestVaultAddress());
    manager.transferNoTimelockManager(noTimelockManager);
    manager.transferAddressTimelockManager(addressTimelockManager);
}
```

Listing 7.2 The transferring admins process of *FwxFactoryLogic* contract

No. 8	**Potential Denial Of Service On APHPool**		
Risk	**Critical**	**Likelihood**	**High**
		Impact	**High**
Functionality is in use	**In use**	**Status**	**Fixed**
Associated Files	*contracts/src/pool/logic/PoolBaseFunc.sol*		
Locations	*PoolBaseFunc. _getActualTokenPrice function L: 105 - 111* *PoolBaseFunc._getInterestTokenPrice function L: 113 - 122* *All the functions that use these two functions*		

Detailed Issue

We found that there is **no handling of the case where the *atpPrice* and *itpPrice* are returned as 0** from the ***_getActualTokenPrice*** and ***_getInterestTokenPrice*** functions (code snippet 8.2), respectively.

This vulnerability allows the potential for **denial of service attack,** as a **division by zero reverts** on the crucial parts of *APHPool* protocol, making the related ***APHPool*** and the ***APHCore* market of the attacked *APHPools* unusable**.

PoolBaseFunc.sol

```
105  function _getActualTokenPrice() internal view returns (uint256) {
106      if (atpTokenTotalSupply == 0) {
107          return initialAtpPrice;
108      } else {
109          return ((pTokenTotalSupply - loss) * PRECISION_UNIT) /
     atpTokenTotalSupply;
110      }
111  }
112
113  function _getInterestTokenPrice() internal view returns (uint256) {
114      if (itpTokenTotalSupply == 0) {
115          return initialItpPrice;
116      } else {
117          return
118              ((pTokenTotalSupply +
119                  IInterestVault(interestVaultAddress).claimableTokenInterest()) *
120                  PRECISION_UNIT) / itpTokenTotalSupply;
121      }
122  }
```

Listing 8.1 The *_getActualTokenPrice* and *_getInterestTokenPrice* functions of the *PoolBaseFunc* contract

The affected functions are as follows:

- The internal ***_deposit*** function
- The internal ***_withdraw*** function
- The internal ***_claimTokenInterest*** function
- The internal ***_activateRank*** function
- All the external functions that used these listed functions

The **example affected functions** are shown below:

PoolLending.sol

```solidity
function _deposit(
    address receiver,
    uint256 nftId,
    uint256 depositAmount
) internal returns (uint256 pMintAmount, uint256 atpMintAmount, uint256 itpMintAmount) {
    require(depositAmount > 0, "PoolLending/deposit-amount-is-zero");

    uint256 atpPrice = _getActualTokenPrice();
    uint256 itpPrice = _getInterestTokenPrice();

    //mint ip, atp, itp
    pMintAmount = _mintPToken(receiver, nftId, depositAmount);

    atpMintAmount = _mintAtpToken(
        receiver,
        nftId,
        ((depositAmount * PRECISION_UNIT) / atpPrice),
        atpPrice
    );

    itpMintAmount = _mintItpToken(
        receiver,
        nftId,
        ((depositAmount * PRECISION_UNIT) / itpPrice),
        itpPrice
    );

    emit Deposit(receiver, nftId, depositAmount, pMintAmount, atpMintAmount, itpMintAmount);
}
```

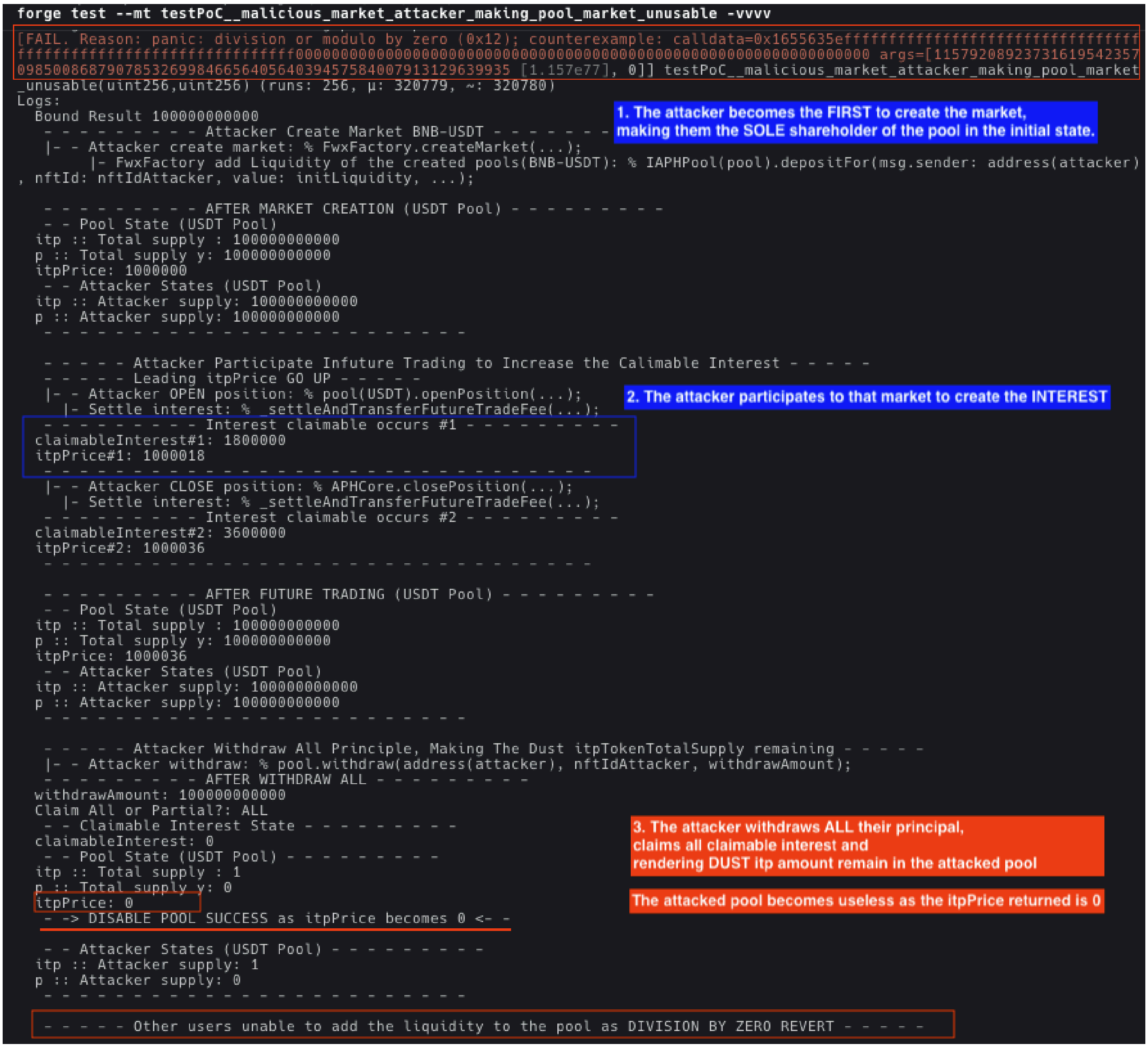
Listing 8.2 The example affected functions

PoolLending.sol

```
function _claimTokenInterest(
    address receiver,
    uint256 nftId,
    uint256 claimAmount
) internal returns (WithdrawResult memory result) {
    uint256 itpPrice = _getInterestTokenPrice();
    PoolTokens storage tokenHolder = tokenHolders[nftId];

    uint256 claimableAmount;
    if (((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) > tokenHolder.pToken) {
        claimableAmount =
            ((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) -
            tokenHolder.pToken;
    }

    claimAmount = MathUpgradeable.min(claimAmount, claimableAmount);

    uint256 burnAmount = _burnItpToken(
        receiver,
        nftId,
        (claimAmount * PRECISION_UNIT) / itpPrice,
        itpPrice
    );
    uint256 bonusAmount = (claimAmount * _getPoolRankInfo(nftId).interestBonusLending) /
        WEI_PERCENT_UNIT;

    uint256 feeSpread = IAPHCore(coreAddress).feeSpread();
    uint256 profitAmount = ((claimAmount * feeSpread) / (WEI_PERCENT_UNIT - feeSpread)) -
        bonusAmount;

    IInterestVault(interestVaultAddress).withdrawTokenInterest(
        claimAmount,
        bonusAmount,
        profitAmount
    );

    emit ClaimTokenInterest(receiver, nftId, claimAmount, bonusAmount, burnAmount);

    result.tokenInterest = claimAmount;
    result.itpTokenBurn = burnAmount;
    result.tokenInterestBonus = bonusAmount;
}
```

Listing 8.3 The example affected functions

The **proof of concept** of the attack scenario at the early stage after its creation is as follows:



```
forge test --mt testPoC__malicious_market_attacker_making_pool_market_unusable -vvvv
[FAIL. Reason: panic: division or modulo by zero (0x12); counterexample: calldata=0x1655635efffffffffffffffffffffffffffffffff
fffffffffffffffffffffffffffffff0000000000000000000000000000000000000000000000000000000000000000 args=[1157920892373161954235
7098500868790785326998466564056403945758400791312963993 5 [1.157e77], 0]] testPoC__malicious_market_attacker_making_pool_market
_unusable(uint256,uint256) (runs: 256, μ: 320779, ~: 320780)
Logs:
  Bound Result 100000000000
   - - - - - - - - - Attacker Create Market BNB-USDT - - - - - - -
   |- - Attacker create market: % FwxFactory.createMarket(...);
        |- FwxFactory add Liquidity of the created pools(BNB-USDT): % IAPHPool(pool).depositFor(msg.sender: address(attacker)
, nftId: nftIdAttacker, value: initLiquidity, ...);

   - - - - - - - - - AFTER MARKET CREATION (USDT Pool) - - - - - - - - -
   - - Pool State (USDT Pool)
  itp :: Total supply : 100000000000
  p :: Total supply y: 100000000000
  itpPrice: 1000000
   - - Attacker States (USDT Pool)
  itp :: Attacker supply: 100000000000
  p :: Attacker supply: 100000000000
   - - - - - - - - - - - - - - - - - - - - - - - -

   - - - - - Attacker Participate Infuture Trading to Increase the Calimable Interest - - - - -
   - - - - - Leading itpPrice GO UP - - - - -
   |- - Attacker OPEN position: % pool(USDT).openPosition(...);
     |- Settle interest: % _settleAndTransferFutureTradeFee(...);
   - - - - - - - - - Interest claimable occurs #1 - - - - - - - - -
  claimableInterest#1: 1800000
  itpPrice#1: 1000018
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   |- - Attacker CLOSE position: % APHCore.closePosition(...);
     |- Settle interest: % _settleAndTransferFutureTradeFee(...);
   - - - - - - - - - Interest claimable occurs #2 - - - - - - - - -
  claimableInterest#2: 3600000
  itpPrice#2: 1000036
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

   - - - - - - - - - AFTER FUTURE TRADING (USDT Pool) - - - - - - - - -
   - - Pool State (USDT Pool)
  itp :: Total supply : 100000000000
  p :: Total supply y: 100000000000
  itpPrice: 1000036
   - - Attacker States (USDT Pool)
  itp :: Attacker supply: 100000000000
  p :: Attacker supply: 100000000000
   - - - - - - - - - - - - - - - - - - - - - - - -

   - - - - - Attacker Withdraw All Principle, Making The Dust itpTokenTotalSupply remaining - - - - -
   |- - Attacker withdraw: % pool.withdraw(address(attacker), nftIdAttacker, withdrawAmount);
   - - - - - - - - - AFTER WITHDRAW ALL - - - - - - - - -
  withdrawAmount: 100000000000
  Claim All or Partial?: ALL
   - - Claimable Interest State - - - - - - - - -
  claimableInterest: 0
   - - Pool State (USDT Pool) - - - - - - - - -
  itp :: Total supply : 1
  p :: Total supply y: 0
  itpPrice: 0
   - -> DISABLE POOL SUCCESS as itpPrice becomes 0 <- -

   - - Attacker States (USDT Pool) - - - - - - - - -
  itp :: Attacker supply: 1
  p :: Attacker supply: 0
   - - - - - - - - - - - - - - - - - - - - - - - -

   - - - - - Other users unable to add the liquidity to the pool as DIVISION BY ZERO REVERT - - - - -
```

1. The attacker becomes the FIRST to create the market, making them the SOLE shareholder of the pool in the initial state.

2. The attacker participates to that market to create the INTEREST

3. The attacker withdraws ALL their principal, claims all claimable interest and rendering DUST itp amount remain in the attacked pool

The attacked pool becomes useless as the itpPrice returned is 0

Listing 8.4 The PoC of this issue

Recommendations

We recommend implementing **handling for cases where the *_getActualTokenPrice* and *_getInterestTokenPrice* functions return 0**.

Reassessment

The *FWX* team prevents the *_getActualTokenPrice* and *_getInterestTokenPrice* functions from returning 0 by burning all *atpToken* and *itpToken* when there is no *pToken* left.

This mitigation ensures that the dust amount caused by arithmetic will be cleared during both the holder withdrawal process and when the APH pool has no principle left.

This fix is shown in the Listing 8.5.

PoolLending.sol

```solidity
function _withdraw(
    address receiver,
    uint256 nftId,
    uint256 withdrawAmount
) internal returns (WithdrawResult memory) {

    // (...SNIPPED...)

    // burn dust token after withdraw all principal
    if (tokenHolder.pToken == 0) {
           if (tokenHolder.atpToken > 0)
               atpBurnAmount += _burnAtpToken(address(this), nftId, tokenHolder.atpToken, 0);
           if (tokenHolder.itpToken > 0)
               itpBurnAmount += _burnItpToken(address(this), nftId, tokenHolder.itpToken, 0);
    }
    // burn total supply when pool is empty
    if (pTokenTotalSupply == 0) {
        loss = 0;
        itpTokenTotalSupply = 0;
        atpTokenTotalSupply = 0;
    }
```

Listing 8.5 The improved *_withdraw* function of the *PoolLending* contract

No. 9	Loss Tracking Precision Mismatch In APHCore		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/APHCore.sol*		
Locations	*APHCore.addLossInUSD L: 74 - 86*		

Detailed Issue

We found that **the implementation of the *addLossInUSD* function of the *APHCore* contract does not support the multiple token precisions.**

To elaborate, The result of the ***lossAmount = (lossAmount * rate) / WEI_UNIT;* will return the *lossAmount* in the precision of itself** as

- ***rate*** is represented by the **18** precisions
- ***WEI_UNIT*** is represented by the **18** precisions
- ***lossAmount*** is represented **according to the *APHPool* token precision**

The precision of the ***lossAmount*** is ***APHPool_token_precision + 18 - 18 = APHPool_token_precision***

Given that the *APHCore* contract can interact with multiple *APHPool* contracts, each of which may involve different precision levels in the amount calculation based on the pool's token precision, we consider the scenarios where *APHCore* contract interacts with *APHPool* contracts that have varying precision.

As a result, the ***nftsLossInUSD[nftId]*** and ***totalLossInUSD*** values become **inaccurate due to the mixing of the precision amounts of each incoming *lossAmount* from the different *APHPool* contracts.**

PoolLending.sol

```
function _withdraw(
    address receiver,
    uint256 nftId,
    uint256 withdrawAmount
) internal returns (WithdrawResult memory) {

    // (...SNIPPED...)

    uint256 lossBurnAmount = withdrawAmount - actualWithdrawAmount;
    loss -= lossBurnAmount;

    IAPHCore(coreAddress).addLossInUSD(nftId, lossBurnAmount);

    // (...SNIPPED...)

}
```

Listing 9.1 The *_withdraw* function of the *PoolLending* contract

APHCore.sol

```
function addLossInUSD(uint256 nftId, uint256 lossAmount) external {
    require(poolToAsset[msg.sender] != address(0), "APHCore/caller-is-not-pool");

    uint256 rate;
    {
        (rate, ) = _queryRateUSD(IAPHPool(msg.sender).tokenAddress());
    }
    lossAmount = (lossAmount * rate) / WEI_UNIT;
    nftsLossInUSD[nftId] = nftsLossInUSD[nftId] + lossAmount;
    totalLossInUSD = totalLossInUSD + lossAmount;

    emit AddLossInUSD(address(this), msg.sender, nftId, lossAmount);
}
```

Listing 9.2 The *addLossInUSD* function of the *APHCore* contract

Recommendations

We recommend updating the formula to support the multiple precisions of incoming ***lossAmount*** as shown below.

The formula recommendation:

lossAmount = (lossAmount * rate) / tokenPrecisionUnit[poolToAsset[msg.sender]];

will return the *lossAmount* in the precision of 18 as

- ***rate*** is represented by the **18** precisions
- ***tokenPrecisionUnit[poolToAsset[msg.sender]]*** is represented **according to the *APHPool* token precision**
    - ***poolToAsset[msg.sender]*** returns the address of the *APHPool* caller's underlying/token address.
- ***lossAmount*** is represented **according to the *APHPool* token precision**.

The result precision of the ***lossAmount*** is

APHPool_token_precision + 18 - APHPool_token_precision = 18

APHCore.sol

```
function addLossInUSD(uint256 nftId, uint256 lossAmount) external {
    require(poolToAsset[msg.sender] != address(0), "APHCore/caller-is-not-pool");

    uint256 rate;
    {
        (rate, ) = _queryRateUSD(IAPHPool(msg.sender).tokenAddress());
    }
    lossAmount = (lossAmount * rate) /
tokenPrecisionUnit[poolToAsset[msg.sender]];
    nftsLossInUSD[nftId] = nftsLossInUSD[nftId] + lossAmount;
    totalLossInUSD = totalLossInUSD + lossAmount;

    emit AddLossInUSD(address(this), msg.sender, nftId, lossAmount);
}
```

Listing 9.3 The improved *addLossInUSD* function of the *APHCore* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

In the reassessment process, the *FWX* team has acknowledged and decided to remove the *addLossInUSD* function. This decision resolves this issue, and the status of this issue can be marked as ***Fixed***.

No. 10	Incorrect Margin Position Validation For Collateral Withdrawal		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/logic/CoreFutureWallet.sol*		
Locations	*CoreFutureWallet._withdrawCollateral L: 93 - 146*		

Detailed Issue

The protocol allows withdrawing collateral only if the position margin is greater or equal to the minimum margin (20%).

However, we found that the *_withdrawCollateral* function of the *CoreFutureWallet* contract has the incorrect condition to validate the margin at the end of the withdrawal process (L130 - 131 in code snippet 10.1), **resulting in the user cannot withdraw collateral when the position margin is exactly equal to the minimum margin.**

CoreFutureWallet.sol

```
 93  function _withdrawCollateral(
 94      uint256 nftId,
 95      address collateralTokenAddress,
 96      address underlyingTokenAddress,
 97      uint256 amount
 98  ) internal returns (uint256) {

         // (...SNIPPED...)

128      require(
129          pos.id == 0 ||
130              _getPositionMargin(nftId, pairByte, true, false) >
131              positionConfigs[pairByte].minimumMargin,
132          "CoreTrading/margin-too-low"
133      );

         // (...SNIPPED...)
147  }
```

Listing 10.1 The *_withdrawCollateral* function of the *CoreFutureWallet* contract

CoreFutureOpening.sol

```
function _updateWalletAndValidateMarginForOpeningPosition(
    OpenedPositionReturn memory openPos,
    APHLibrary.OpenPositionParams memory params,
    bytes32 pairByte,
    uint256 wallet
) internal {
    require(wallet >= (openPos.collaUsed + openPos.swapFee), "CoreTrading/wallet-insuficient");
    wallet = _updateWallet(
        params.nftId,
        pairByte,
        wallet - openPos.collaUsed - openPos.swapFee
    );

    // force to use router 1 for bypassing oracle checking
    uint256 margin = _getPositionMargin(params.nftId, pairByte, false, true);
    require(margin >= positionConfigs[pairByte].minimumMargin, "CoreTrading/margin-too-low");
}
```

Listing 10.2 The *_updateWalletAndValidateMarginForOpeningPosition* function of the *CoreFutureOpening* contract

Recommendations

We recommend re-implementing the mentioned condition as shown in the code snippet below.

CoreFutureWallet.sol

```
function _withdrawCollateral(
    uint256 nftId,
    address collateralTokenAddress,
    address underlyingTokenAddress,
    uint256 amount
) internal returns (uint256) {

    // (...SNIPPED...)

    require(
        pos.id == 0 ||
            _getPositionMargin(nftId, pairByte, true, false) >=
            positionConfigs[pairByte].minimumMargin,
        "CoreTrading/margin-too-low"
    );

    // (...SNIPPED...)
}
```

Listing 10.3 The improved *_withdrawCollateral* function of the *CoreFutureWallet* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

CoreFutureWallet.sol

```
// (...SNIPPED...)

Position memory pos = positions[nftId][pairByte];
require(
    pos.id == 0 ||
        _getPositionMargin(nftId, pairByte, true, false) >=
        positionConfigs[pairByte].minimumMargin,
    "CoreTrading/margin-too-low"
);
```

Listing 10.4 The improved *_withdrawCollateral* function of the *CoreFutureWallet* contract

No. 11	Lock Of The Borrow Token In The APHCore Due To Double Subtracted Fee		
Risk	Critical	Likelihood	High
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/logic/CoreFutureClosing.sol*		
Locations	*CoreFutureClosing._closeLong L: 279 and 316*		

Detailed Issue

We found that **the *_closeLong* function of the *CoreFutureClosing* contract double subtracted the *swapFee* amount from the *actualCollateral* value when the execution fell into the Loss case**.

From the code shown below, the *result.swapFee* has already been subtracted (L279 in code snippet 11.1) from the *amounts[1]* (amount output of the swap that is not included swap fee) before adding to the *actualCollateral* variable.

However, **in the case that loss occurs, the *actualCollateral* variable that already collects the actual collateral including the *result.swapFee*, has been subtracted by the *result.swapFee* again before assigning to the *result.repayAmount* L316 in code snippet 11.1** making the **incorrect** *result.repayAmount* value to transfer back to the associated *APHPool*.

The ***result.repayAmount*** will contain the incorrect value as

actualCollateral = *actualCollateral + amounts[1] - result.swapFee; L279*

result.repayAmount = *actualCollateral* **- *result.swapFee***; *L316*

= *actualCollateral + amounts[1] - result.swapFee* **- *result.swapFee;***

As a result, the ***result.repayAmount*** can be incorrect as double subtraction and after that it is transferred to the associated APHPool, making some funds from double subtraction locked in the *APHCore* contract.

Expected: ***result.repayAmount = (actualCollateral + amounts[1] - result.swapFee)***

Actual: ***result.repayAmount = (actualCollateral + amounts[1] - result.swapFee) - result.swapFee***

The locked value in the *APHCore* contract for each closing with a loss will be ***result.swapFee***

CoreFutureClosing.sol

```
function _closeLong(
        APHLibrary.ClosePositionParams memory params
    ) internal returns (APHLibrary.ClosePositionResponse memory result) {
        Pair memory pair = pairs[params.pairByte];
        PoolStat storage poolStat = poolStats[assetToPool[pair.pair0]];
        Position storage pos = positions[params.nftId][params.pairByte];
        PositionState storage posState =
positionStates[params.nftId][params.posId];

        // (...SNIPPED...)

        // calculate fee
        result.tradingFee = _getFeeAmount(amounts[1], params.tradingFee);
        result.repayAmount = (params.closingSize * pos.borrowAmount) /
pos.contractSize;

        // calculate real actualCollateral
        actualCollateral = actualCollateral + amounts[1] - result.swapFee;
        bool isCritical = actualCollateral < result.repayAmount;

        if (isCritical == false) {
            actualCollateral -= result.repayAmount;
            (actualCollateral, result.tradingFee) = _cascadeActualCollateral(
                pos,
                posState,
                actualCollateral,
                result.tradingFee
            );

            _updateWallet(params.nftId, params.pairByte, actualCollateral);

            uint256 newInterestOwedPerDay = (pos.interestOwePerDay *
                (pos.contractSize - params.closingSize)) / pos.contractSize;
            uint256 collateralSwappedAmountReturn = MathUpgradeable.min(
                (pos.collateralSwappedAmount * params.closingSize) /
pos.contractSize,
                pos.collateralSwappedAmount
            );

            // update pool stat
            poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
            poolStat.borrowInterestOwedPerDayFromTrading -=
(pos.interestOwePerDay -
                newInterestOwedPerDay);

            // update position
            pos.borrowAmount -= result.repayAmount;
            pos.collateralSwappedAmount -= collateralSwappedAmountReturn;
            pos.interestOwePerDay = newInterestOwedPerDay;
            pos.contractSize -= params.closingSize;
        } else {
            // ! LOSS
            poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
            poolStat.borrowInterestOwedPerDayFromTrading -=
pos.interestOwePerDay;

            IAPHPool(assetToPool[pair.pair0]).addLoss(result.repayAmount -
actualCollateral);
            result.repayAmount = actualCollateral - result.swapFee;

            _updateWallet(params.nftId, params.pairByte, 0);

            pos.contractSize = 0;
            result.tradingFee = 0;
        }

        uint256 lenderFeeAmount = _getFeeAmount(result.tradingFee,
tradingFeeToLender);
        result.feeToProfitVault = result.tradingFee - lenderFeeAmount;
        result.feeToIntVault = lenderFeeAmount + (posState.interestPaid -
interestPaid);
        result.pnl = APHLibrary._calculatePNL(
            result.rate,
            pos.entryPrice,
            params.closingSize,
            underlyingPrecision
        );

        poolStat.totalInterestPaidFromTrading += (result.feeToIntVault);

        pos.interestOwed = pos.interestOwed - (posState.interestPaid -
interestPaid);

        posState.PNL += result.pnl;
        posState.totalSwapFee += uint128(result.swapFee);
        posState.totalTradingFee += uint128(result.tradingFee);

        if (pos.contractSize == 0) _resetPosition(params.nftId, pos.id,
params.pairByte);
}
```

Listing 11.1 The *_closeLong* function of the *CoreFutureClosing* contract

CoreFutureClosing.sol

```
function _closePosition(uint256 nftId, uint256 _posId, uint256 _closingSize) internal {
     require(_closingSize != 0, "CoreTrading/closingSize-is-zero");
     require(_posId != 0, "CoreTrading/posId-is-zero");
     // (...SNIPPED...)

     // repay borrowing tokens back to pool.
     _safeTransfer(
           posState.isLong ? pair.pair0 : pair.pair1,
           assetToPool[posState.isLong ? pair.pair0 : pair.pair1],
           result.repayAmount
     );

     // (...SNIPPED...)
}
```

Listing 11.2 The *_closePosition* function of the *CoreFutureClosing* contract

Recommendations

We recommend updating the incorrect calculation code as follows:

CoreFutureClosing.sol

```
if (isCritical == false) {
         // (...SNIPPED...)
     } else {
         // ! LOSS
         poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
         poolStat.borrowInterestOwedPerDayFromTrading -= pos.interestOwePerDay;

         IAPHPool(assetToPool[pair.pair0]).addLoss(result.repayAmount - actualCollateral);
         result.repayAmount = actualCollateral;

         _updateWallet(params.nftId, params.pairByte, 0);

         pos.contractSize = 0;
         result.tradingFee = 0;
}
```

Listing 11.3 The improved *_closeLong* function of the *CoreFutureClosing* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 12	Loss Of Claimable Interest In Rounding Down Issue		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/pool/logic/PoolBaseFunc.sol* *contracts/src/pool/logic/PoolLending.sol*		
Locations	*PoolBaseFunc._getInterestTokenPrice L: 113 - 122* *PoolLending._claimTokenInterest L: 252 - 293*		

Detailed Issue

We found the potential **rounding down issues** in the ***itpPrice*** calculation that affect the ***_claimTokenInterest*** function (code snippet 12.1). **This issue creates a loss of interest claimable for shareholders in the *APHPool*** and any remaining claimable interest from the rounding down issue will be shared among other participating lenders.

Although the potential loss value is negligible in comparison to each precision of the calculated value, we are concerned that the actual loss value will depend on the real value of the token, such as its price.

However, the case of loss due to rounding down is the limitation of Solidity as ***Solidity*** has not fully supported the fixed point numbers yet and cannot define the precise decimal representation.

As a result, careful consideration and additional loss-tracking mechanisms are necessary to mitigate potential discrepancies in cases where rounding down might impact the accuracy of calculations.

PoolLending.sol

```
252 function _claimTokenInterest(
253     address receiver,
254     uint256 nftId,
255     uint256 claimAmount
256 ) internal returns (WithdrawResult memory result) {
257     uint256 itpPrice = _getInterestTokenPrice();
258     PoolTokens storage tokenHolder = tokenHolders[nftId];
259
260     uint256 claimableAmount;
261     if (((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) >
    tokenHolder.pToken) {
262         claimableAmount =
263             ((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) -
```

```
            tokenHolder.pToken;
    }

    claimAmount = MathUpgradeable.min(claimAmount, claimableAmount);

    uint256 burnAmount = _burnItpToken(
        receiver,
        nftId,
        (claimAmount * PRECISION_UNIT) / itpPrice,
        itpPrice
    );
    uint256 bonusAmount = (claimAmount *
_getPoolRankInfo(nftId).interestBonusLending) /
        WEI_PERCENT_UNIT;

    uint256 feeSpread = IAPHCore(coreAddress).feeSpread();
    uint256 profitAmount = ((claimAmount * feeSpread) / (WEI_PERCENT_UNIT -
feeSpread)) -
        bonusAmount;

    IInterestVault(interestVaultAddress).withdrawTokenInterest(
        claimAmount,
        bonusAmount,
        profitAmount
    );

    emit ClaimTokenInterest(receiver, nftId, claimAmount, bonusAmount,
burnAmount);

    result.tokenInterest = claimAmount;
    result.itpTokenBurn = burnAmount;
    result.tokenInterestBonus = bonusAmount;
}
```

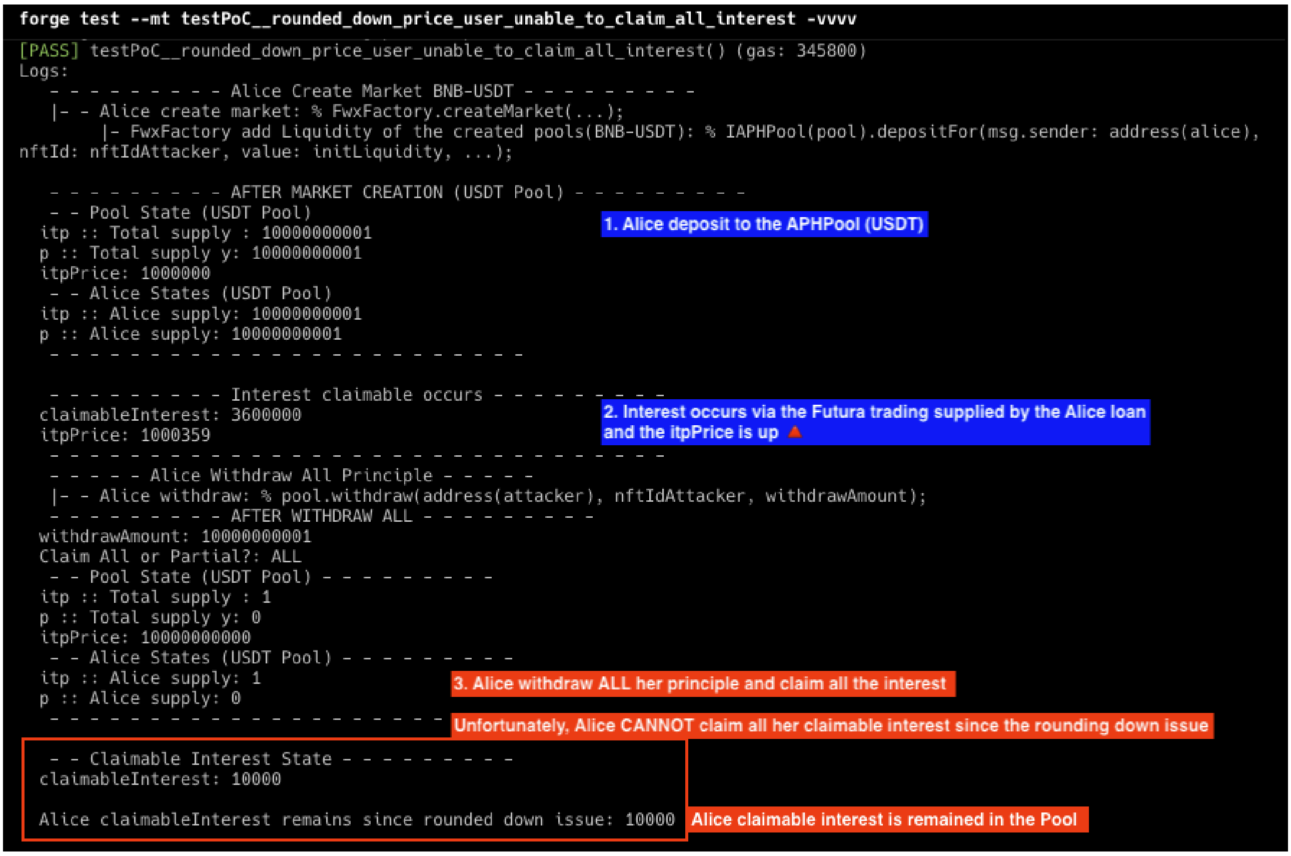
Listing 12.1 The *_claimTokenInterest* function of the *PoolLending* contract

PoolLending.sol

```
function _getInterestTokenPrice() internal view returns (uint256) {
    if (itpTokenTotalSupply == 0) {
        return initialItpPrice;
    } else {
        return
            ((pTokenTotalSupply +
                IInterestVault(interestVaultAddress).claimableTokenInterest()) *
                PRECISION_UNIT) / itpTokenTotalSupply;
    }
}
```

Listing 12.2 The *_getInterestTokenPrice* function of the *PoolBaseFunc* contract

The **proof of concept** of the unfair scenario is as follows:



```
forge test --mt testPoC__rounded_down_price_user_unable_to_claim_all_interest -vvvv
[PASS] testPoC__rounded_down_price_user_unable_to_claim_all_interest() (gas: 345800)
Logs:
   - - - - - - - - - Alice Create Market BNB-USDT - - - - - - - - -
    |- - Alice create market: % FwxFactory.createMarket(...);
         |- FwxFactory add Liquidity of the created pools(BNB-USDT): % IAPHPool(pool).depositFor(msg.sender: address(alice),
nftId: nftIdAttacker, value: initLiquidity, ...);

   - - - - - - - - - AFTER MARKET CREATION (USDT Pool) - - - - - - - - -
   - - Pool State (USDT Pool)
  itp :: Total supply : 10000000001
  p :: Total supply y: 10000000001
  itpPrice: 1000000
   - - Alice States (USDT Pool)
  itp :: Alice supply: 10000000001
  p :: Alice supply: 10000000001
   - - - - - - - - - - - - - - - - - - - - - - - -

   - - - - - - - - - Interest claimable occurs - - - - - - - - -
  claimableInterest: 3600000
  itpPrice: 1000359
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   - - - - - Alice Withdraw All Principle - - - - -
   |- - Alice withdraw: % pool.withdraw(address(attacker), nftIdAttacker, withdrawAmount);
   - - - - - - - - - AFTER WITHDRAW ALL - - - - - - - - -
  withdrawAmount: 10000000001
  Claim All or Partial?: ALL
   - - Pool State (USDT Pool) - - - - - - - - -
  itp :: Total supply : 1
  p :: Total supply y: 0
  itpPrice: 10000000000
   - - Alice States (USDT Pool) - - - - - - - - -
  itp :: Alice supply: 1
  p :: Alice supply: 0
   - - - - - - - - - - - - - - - - - - - -
   - - Claimable Interest State - - - - - - - - -
  claimableInterest: 10000

  Alice claimableInterest remains since rounded down issue: 10000
```

1. Alice deposit to the APHPool (USDT)

2. Interest occurs via the Futura trading supplied by the Alice loan and the itpPrice is up ▲

3. Alice withdraw ALL her principle and claim all the interest

Unfortunately, Alice CANNOT claim all her claimable interest since the rounding down issue

Alice claimable interest is remained in the Pool

Listing 12.3 The proof of concept of the unfair scenario

Recommendations

As the case of loss due to rounding down is the limitation of ***Solidity*** as ***Solidity*** has not fully supported the fixed point numbers yet and cannot define the precise decimal representation.

Consequently, careful consideration and additional loss-tracking mechanisms are necessary to mitigate potential discrepancies in cases where rounding down might impact the accuracy of calculations.

Reassessment

The *FWX* team has acknowledged this issue with the statement:

“The fix of the issue ‘Potential Denial Of Service On APHPool’ will reset users’ balances to zero. The left-over claimableInterest will be compounded for future lenders.”

No. 13	Potential Of Global Setting Precisions Mismatch With Token Precisions		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/factory/logic/FwxFactoryLogic.sol* *contracts/src/core/logic/CoreFutureOpening.sol*		
Locations	*FwxFactoryLogic._setupConfigs L: 216 - 217* *CoreFutureOpening._verifyOpeningPositionSize L: 299 - 313*		

Detailed Issue

We found that the potential of precision mismatch between the global settings *minOpenPositionSize* and *maxOpenPositionSize* (L216 - 217 in code snippet 13.1) and the token precisions in the *_verifyOpeningPositionSize* function (L299 - 313 in code snippet 13.2). This discrepancy may lead to inaccuracies when calculating position sizes.

To illustrate, **consider a scenario where *minOpenPositionSize* is configured at 100e18 and *maxOpenPositionSize* is set to 1000e18**. In this setup, the intention is to enable a minimum position size of 100, ensuring that the overall value doesn't surpass 1000 for tokens with 18 decimals. **However, issues may arise with unintended position sizes when dealing with tokens of varying decimal precision.**

Specifically, for tokens with 6 decimals, the same configuration allows a minimum position size of 100e12, with the total value not exceeding 1000e12.

FwxFactoryLogic.sol

```
    contract FwxFactoryLogic is FwxFactoryBase, FwxFactoryProxyBase,
    IFwxFactoryLogic {

        // (...SNIPPED...)

166     function _setupConfigs(
167         address core,
168         address collateralPool,
169         address underlyingPool,
170         address collateralToken,
171         address underlyingToken
172     ) internal {
173         IAPHCoreSetting coreSetting = IAPHCoreSetting(core);
174
```

```
        /* ---------------------------------- pool ---------------------------------- */
        coreSetting.registerNewPool(collateralPool);
        coreSetting.registerNewPool(underlyingPool);

        // (...SNIPPED...)

        coreSetting.setPositionConfig(
            collateralToken,
            underlyingToken,
            positionCfg.maintenanceMargin,
            positionCfg.minimumMargin,
            positionCfg.bountyFeeRateToProtocol,
            positionCfg.bountyFeeRateToLiquidator,
            positionCfg.minOpenPositionSize,
            positionCfg.maxOpenPositionSize
        );
        coreSetting.approveForRouter(collateralToken, 0, type(uint256).max);
        coreSetting.approveForRouter(underlyingToken, 0, type(uint256).max);
    }

    // (...SNIPPED...)
}
```

Listing 13.1 The *_setupConfigs* function of the *FwxFactoryLogic* contract

CoreFutureOpening.sol

```
contract CoreFutureOpening is CoreFutureBaseFunc {
    // (...SNIPPED...)

    function _verifyOpeningPositionSize(
        PositionConfig memory config,
        uint256 newPositionValue,
        uint256 totalPositionValue
    ) internal pure {
        require(
            newPositionValue >= config.minOpenPositionSize,
            "CoreTrading/position-size-is-too-small"
        );

        require(
            totalPositionValue <= config.maxOpenPositionSize,
            "CoreTrading/position-size-is-too-big"
        );
    }

    // (...SNIPPED...)
}
```

Listing 13.2 The _*verifyOpeningPositionSize* function of the *CoreFutureOpening* contract

Recommendations

The team should confirm whether this aligns with the intended behavior or if there exists a precision mismatch in position size calculations.

In case of a precision mismatch, the team should ensure proper alignment of global settings with tokens of varying decimals.

Reassessment

The *FWX* team has acknowledged this issue and the team will set the variables ***minOpenPositionSize*** and ***maxOpenPositionSize*** to ***zero*** and ***type(uint).max***, respectively.

No. 14	Potential Inability To Withdraw Principal Token Due To Arithmetic Underflow Revert		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/logic/PoolBaseFunc.sol* *contracts/src/pool/logic/PoolLending.sol*		
Locations	*PoolBaseFunc._getActualTokenPrice L: 105 - 111* *PoolLending._withdraw L: 183 - 238*		

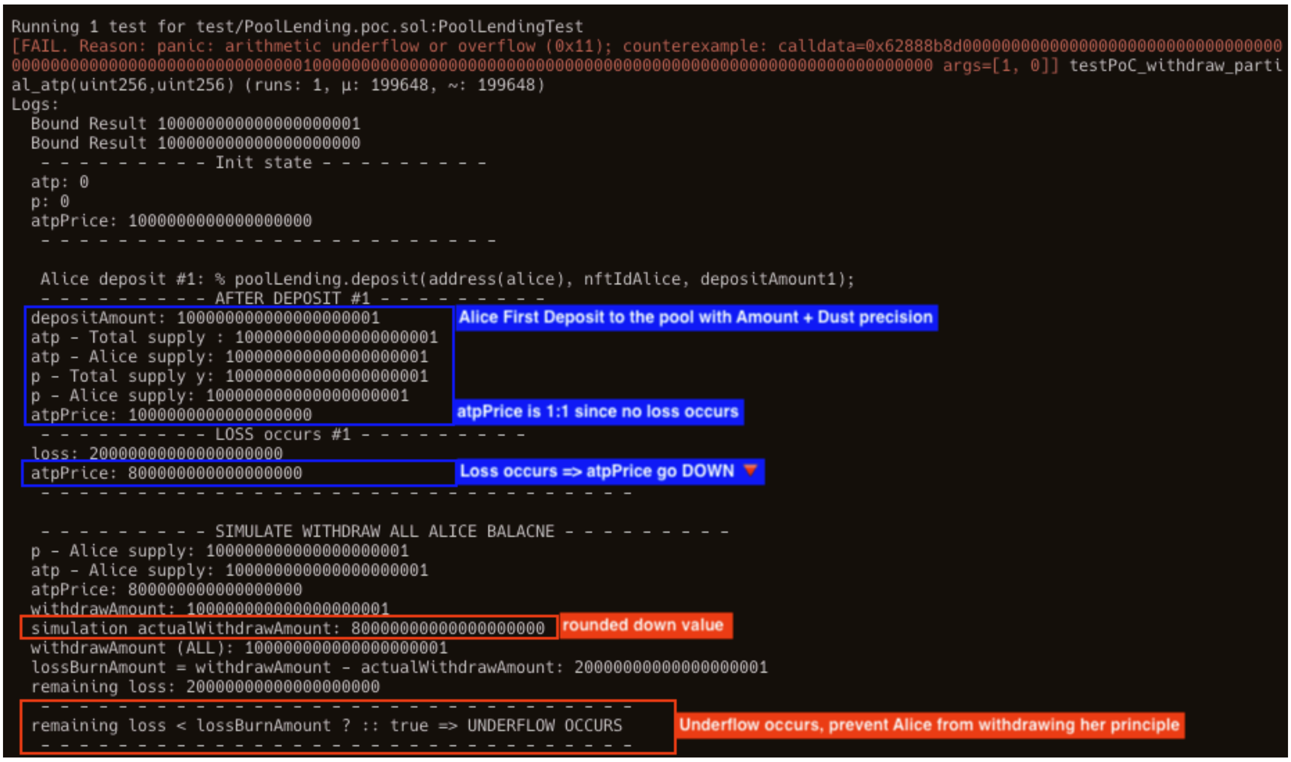
Detailed Issue

We found the potential **underflow reverts** due to the **rounding down of *atpPrice*** and ***actualWithdrawAmount*** calculations that affect the ***_withdraw*** function. **This issue potentially prevents lenders from withdrawing their principal**.

To elaborate, the **rounding down in the calculation of *atpPrice*** and ***actualWithdrawAmount*** potentially results in the ***lossBurnAmount*** value being greater than the total ***loss*** tracked of the APHPool.

As a result, the **underflow revert occurs preventing lenders from withdrawing their principal.**

The **proof of concept** of the issue scenario is as follows:



```
Running 1 test for test/PoolLending.poc.sol:PoolLendingTest
[FAIL. Reason: panic: arithmetic underflow or overflow (0x11); counterexample: calldata=0x62888b8d0000000000000000000000000000000
000000000000000000000000000000001000000000000000000000000000000000000000000000000000000000000000000 args=[1, 0]] testPoC_withdraw_parti
al_atp(uint256,uint256) (runs: 1, μ: 199648, ~: 199648)
Logs:
  Bound Result 100000000000000000001
  Bound Result 100000000000000000000
   - - - - - - - - - Init state - - - - - - - - -
  atp: 0
  p: 0
  atpPrice: 1000000000000000000
   - - - - - - - - - - - - - - - - - - - - - - - -

   Alice deposit #1: % poolLending.deposit(address(alice), nftIdAlice, depositAmount1);
   - - - - - - - - - AFTER DEPOSIT #1 - - - - - - - - -
  depositAmount: 100000000000000000001
  atp - Total supply : 100000000000000000001
  atp - Alice supply: 100000000000000000001
  p - Total supply y: 100000000000000000001
  p - Alice supply: 100000000000000000001
  atpPrice: 1000000000000000000
   - - - - - - - - - LOSS occurs #1 - - - - - - - - -
  loss: 20000000000000000000
  atpPrice: 800000000000000000
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

   - - - - - - - - - SIMULATE WITHDRAW ALL ALICE BALACNE - - - - - - - - -
  p - Alice supply: 100000000000000000001
  atp - Alice supply: 100000000000000000001
  atpPrice: 800000000000000000
  withdrawAmount: 100000000000000000001
  simulation actualWithdrawAmount: 80000000000000000000
  withdrawAmount (ALL): 100000000000000000001
  lossBurnAmount = withdrawAmount - actualWithdrawAmount: 20000000000000000001
  remaining loss: 20000000000000000000
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  remaining loss < lossBurnAmount ? :: true => UNDERFLOW OCCURS
   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Alice First Deposit to the pool with Amount + Dust precision

atpPrice is 1:1 since no loss occurs

Loss occurs => atpPrice go DOWN 🔻

rounded down value

Underflow occurs, prevent Alice from withdrawing her principle

Listing 14.1 The *Proof of Concept* for the underflow issue

PoolLending.sol

```solidity
function _withdraw(
    address receiver,
    uint256 nftId,
    uint256 withdrawAmount
) internal returns (WithdrawResult memory) {
    PoolTokens storage tokenHolder = tokenHolders[nftId];

    uint256 atpPrice = _getActualTokenPrice();
    uint256 itpPrice = _getInterestTokenPrice();

    // (...SNIPPED...)

    uint256 actualWithdrawAmount = tokenHolder.pToken > 0
        ? MathUpgradeable.min(
            (tokenHolder.atpToken * atpPrice * withdrawAmount) /
                (tokenHolder.pToken * PRECISION_UNIT),
            tokenHolder.pToken
        )
        : 0;

    require(actualWithdrawAmount <= _currentSupply(),
"PoolLending/pool-supply-insufficient");

    // (...SNIPPED...)
    uint256 lossBurnAmount = withdrawAmount - actualWithdrawAmount;
    loss -= lossBurnAmount;

    IAPHCore(coreAddress).addLossInUSD(nftId, lossBurnAmount);

    // (...SNIPPED...)

}
```

Listing 14.2 The *_withdraw* function of the *PoolLending* contract

PoolBaseFunc.sol

```
function _getActualTokenPrice() internal view returns (uint256) {
    if (atpTokenTotalSupply == 0) {
        return initialAtpPrice;
    } else {
        return ((pTokenTotalSupply - loss) * PRECISION_UNIT) / atpTokenTotalSupply;
    }
}
```

Listing 14.3 The *_getActualTokenPrice* function of the *PoolBaseFunc* contract

Recommendations

We recommend implementing a boundary check to handle scenarios where the loss burn amount can be subtracted from the loss variable without triggering arithmetic reverts.

PoolLending.sol

```
function _withdraw(
    address receiver,
    uint256 nftId,
    uint256 withdrawAmount
) internal returns (WithdrawResult memory) {

    // (...SNIPPED...)

    uint256 lossBurnAmount = MathUpgradeable.min(withdrawAmount - actualWithdrawAmount, loss);
    loss -= lossBurnAmount;

    IAPHCore(coreAddress).addLossInUSD(nftId, lossBurnAmount);

    // (...SNIPPED...)
}
```

Listing 14.4 The improved *_withdraw* function of the *PoolLending* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

PoolLending.sol

```
// (...SNIPPED...)

    uint256 actualWithdrawAmount = tokenHolder.pToken > 0
        ? MathUpgradeable.min(
            (tokenHolder.atpToken * atpPrice * withdrawAmount) /
                (tokenHolder.pToken * PRECISION_UNIT),
            tokenHolder.pToken
        )
        : 0;

    require(actualWithdrawAmount <= _currentSupply(),
"PoolLending/pool-supply-insufficient");

    uint256 itpBurnAmount = _burnItpToken(
        receiver,
        nftId,
        (withdrawAmount * PRECISION_UNIT) / itpPrice,
        itpPrice
    );

    uint256 atpBurnAmount = tokenHolder.pToken > 0
        ? ((withdrawAmount * tokenHolder.atpToken) / (tokenHolder.pToken))
        : 0;
    atpBurnAmount = _burnAtpToken(receiver, nftId, atpBurnAmount, atpPrice);

    uint256 pBurnAmount = _burnPToken(receiver, nftId, withdrawAmount);

    uint256 lossBurnAmount = MathUpgradeable.min(withdrawAmount -
actualWithdrawAmount, loss);
    loss -= lossBurnAmount;
```

Listing 14.5 The boundary check of the loss burn amount

No. 15	Potential Rounding Down For SwapFee Calculation		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/logic/CoreSwapping.sol*		
Locations	*CoreSwapping._calculateSwapFee L: 349 - 368*		

Detailed Issue

The *_calculateSwapFee* function of the *CoreSwapping* contract (L359 in code snippet 15.1) may return the rounding down swap fee from the division. Consequently, the protocol may return inaccurate ***swapFee*** and ***resultAmounts*** that affect further calculations.

CoreSwapping.sol

```
349  function _calculateSwapFee(
350      bool isExactOutput,
351      uint256 routerIndex,
352      uint256[] memory amounts,
353      address[] memory path
354  ) internal view returns (uint256[] memory resultAmounts, uint256 swapFee) {
355      resultAmounts = amounts;
356      uint256 swapFeeRate = _getSwapFeeRate(routerIndex, path[0], path[1]);
357
358      if (isExactOutput) {
359          swapFee = (amounts[0] * swapFeeRate) / WEI_PERCENT_UNIT;
360          resultAmounts[0] -= swapFee;
361      } else {
362          (uint256 reserve0, ) = _getReserves(routerIndex, path[0], path[1]);
363          swapFee =
364              (reserve0 * amounts[1] * swapFeeRate) /
365              ((reserve0 + amounts[0]) * (WEI_PERCENT_UNIT - swapFeeRate));
366          resultAmounts[1] += swapFee;
367      }
368  }
```

Listing 15.1 The *_calculateSwapFee* function of the *CoreSwapping* contract

CoreFutureClosing.sol

```
function _closeLong(
    APHLibrary.ClosePositionParams memory params
) internal returns (APHLibrary.ClosePositionResponse memory result) {
    Pair memory pair = pairs[params.pairByte];
    PoolStat storage poolStat = poolStats[assetToPool[pair.pair0]];
    Position storage pos = positions[params.nftId][params.pairByte];
    PositionState storage posState = positionStates[params.nftId][params.posId];

    poolStat.updatedTimestamp = block.timestamp;

    uint256[] memory amounts;
    uint256 interestPaid = posState.interestPaid;
    uint256 actualCollateral = wallets[params.nftId][params.pairByte];

    // swap
    (amounts, result.swapFee, result.router) = params.isLiquidate
        ? _positionLiquidationSwap(
            false,
            params.pairByte,
            params.closingSize,
            1,
            pos.swapTokenAddress,
            pos.borrowTokenAddress,
            address(this)
        )
        : _swap(
            false,
            params.pairByte,
            params.closingSize, // amountIn
            1, // amountOutMin
            pos.swapTokenAddress,
            pos.borrowTokenAddress,
            address(this),
            0,
            0
        );

    uint256 collateralPrecision = tokenPrecisionUnit[pair.pair0];
    uint256 underlyingPrecision = tokenPrecisionUnit[pair.pair1];
    result.rate = (amounts[1] * underlyingPrecision) / amounts[0];
    result.precision = collateralPrecision;

    // calculate fee
    result.tradingFee = _getFeeAmount(amounts[1], params.tradingFee);
    result.repayAmount = (params.closingSize * pos.borrowAmount) /
pos.contractSize;

    // calculate real actualCollateral
    actualCollateral = actualCollateral + amounts[1] - result.swapFee;
    bool isCritical = actualCollateral < result.repayAmount;

    // (...SNIPPED...)
```

Listing 15.2 The example that uses the rounding down result

Recommendations

We recommend adding ***+1*** to the *swapFee* calculation for round-up, making the user pay one more wei to the *APHCore* for sufficient for further calculations.

CoreSwapping.sol

```
function _calculateSwapFee(
    bool isExactOutput,
    uint256 routerIndex,
    uint256[] memory amounts,
    address[] memory path
) internal view returns (uint256[] memory resultAmounts, uint256 swapFee) {
    resultAmounts = amounts;
    uint256 swapFeeRate = _getSwapFeeRate(routerIndex, path[0], path[1]);

    if (isExactOutput) {
        swapFee = (amounts[0] * swapFeeRate) / WEI_PERCENT_UNIT + 1;
        resultAmounts[0] -= swapFee;
    } else {
        (uint256 reserve0, ) = _getReserves(routerIndex, path[0], path[1]);
        swapFee =
            (reserve0 * amounts[1] * swapFeeRate) /
            ((reserve0 + amounts[0]) * (WEI_PERCENT_UNIT - swapFeeRate)) + 1;
        resultAmounts[1] += swapFee;
    }
}
```

Listing 15.3 The improved *_calculateSwapFee* function of the *CoreSwapping* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

PoolLending.sol

```
function _calculateSwapFee(
    bool isExactOutput,
    uint256 routerIndex,
    uint256[] memory amounts,
    address[] memory path
) internal view returns (uint256[] memory resultAmounts, uint256 swapFee) {
    resultAmounts = amounts;
    uint256 swapFeeRate = _getSwapFeeRate(routerIndex, path[0], path[1]);

    if (isExactOutput) {
        swapFee = (amounts[0] * swapFeeRate) / WEI_PERCENT_UNIT + 1;
        resultAmounts[0] -= swapFee;
    } else {
        (uint256 reserve0, ) = _getReserves(routerIndex, path[0], path[1]);
        swapFee =
            (reserve0 * amounts[1] * swapFeeRate) /
            ((reserve0 + amounts[0]) * (WEI_PERCENT_UNIT - swapFeeRate)) +
            1;
        resultAmounts[1] += swapFee;
    }
}
```

Listing 15.4 The improved *_calculateSwapFee* function of the *CoreSwapping* contract

No. 16	Potential Locking Of bountyFeeToLiquidator Within The APHPool		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/logic/PoolBorrowing.sol* *contracts/src/core/logic/CoreFutureClosing.sol*		
Locations	*PoolBorrowing._openPosition L: 42 - 112* *CoreFutureClosing._closePosition L: 24 - 99* *CoreFutureClosing._liquidatePosition L: 106 - 230*		

Detailed Issue

The *APHPool* can perform the liquidation and be the liquidator from opening opposite side position flow.

In the mentioned flow, the *APHPool* will execute liquidation and function as the liquidator. Consequently, the ***bountyFeeToLiquidator*** rewards will be distributed to *APHPool* (the liquidator).

As a result, funds will be permanently locked in the associated *APHPool* contract since there is no implementation to handle this case within the *APHPool* contract.

PoolBorrowing.sol

```
42  function _openPosition(
43          APHLibrary.PoolOpenPositionParams memory poolParams
44      ) internal returns (CoreBase.Position memory pos) {
45          uint256 nftId = _getUsableToken(msg.sender, poolParams.nftId);
46          bytes32 pairByte = APHLibrary._hashPair(
47              poolParams.collateralTokenAddress,
48              poolParams.swapTokenAddress,
49              tokenAddress
50          );
51          pos = IAPHCore(coreAddress).positions(nftId, pairByte);
52
53          bool newIsLong = tokenAddress == poolParams.collateralTokenAddress;
54          uint256 contractSize = poolParams.contractSize;
55
56          // Open new position in opposite side
57          if (pos.id != 0 && pos.borrowTokenAddress != tokenAddress) {
58              uint256 currentContractSize = newIsLong ? pos.borrowAmount :
    pos.contractSize;
59              if (currentContractSize >= contractSize) {
```

```
                IAPHCore(coreAddress).closePosition(nftId, pos.id, contractSize);
                return pos;
            } else {
                IAPHCore(coreAddress).closePosition(nftId, pos.id, currentContractSize);
                contractSize = contractSize - currentContractSize;
            }
        }

// (...SNIPPED...)

}
```

Listing 16.1 The *_openPosition* function of the *PoolBorrowing* contract

CoreFutureClosing.sol

```
function _closePosition(uint256 nftId, uint256 _posId, uint256 _closingSize) internal {
      require(_closingSize != 0, "CoreTrading/closingSize-is-zero");
      require(_posId != 0, "CoreTrading/posId-is-zero");
      // (...SNIPPED...)

      APHLibrary.ClosePositionResponse memory result;
      // close position if current margin is not below maintenanceMagin, otherwise liquidate
      if (
          _getPositionMargin(nftId, posState.pairByte, false, false) >=
          positionConfigs[posState.pairByte].maintenanceMargin
      ) {
          // (...SNIPPED...)
      } else {
          _liquidatePosition(nftId, posState.pairByte);
      }
}
```

Listing 16.2 The *_closePosition* function of the *CoreFutureClosing* contract

CoreFutureClosing.sol

```
function _liquidatePosition(uint256 nftId, bytes32 pairByte) internal {
     Position storage pos = positions[nftId][pairByte];
     PositionState storage posState = positionStates[nftId][pos.id];
     // (...SNIPPED...)

     if (msg.sender != IMembership(membershipAddress).ownerOf(nftId)) {
         // bounty fee
         {
             uint256 wallet = wallets[nftId][pairByte];

             (uint256 rate, ) = _queryRateUSD(tmp.collateralToken);
             uint256 collateralPrecision =
tokenPrecisionUnit[tmp.collateralToken];
             uint256 feeToLiquidator = (liquidationFee * collateralPrecision) /
rate;

             if (feeToLiquidator >= wallet) {
                 feeToLiquidator = wallet;
                 wallet = 0;
             } else {
                 wallet = wallet - feeToLiquidator;

                 tmp.bountyFeeToProtocol =
                        (wallet *
positionConfigs[pairByte].bountyFeeRateToProtocol) /
                        WEI_PERCENT_UNIT;
                 tmp.bountyFeeToLiquidator =
                        (wallet *
positionConfigs[pairByte].bountyFeeRateToLiquidator) /
                        WEI_PERCENT_UNIT;

                 wallet = wallet - tmp.bountyFeeToProtocol -
tmp.bountyFeeToLiquidator;
                }

                tmp.bountyFeeToLiquidator += feeToLiquidator;
                _updateWallet(nftId, pairByte, wallet);
                if (tmp.bountyFeeToLiquidator > 0) {
                    _safeTransfer(tmp.collateralToken, msg.sender,
tmp.bountyFeeToLiquidator);
                }

                if (tmp.bountyFeeToProtocol > 0) {
                    _safeTransfer(tmp.collateralToken, feeVaultAddress,
tmp.bountyFeeToProtocol);
                    IFeeVault(feeVaultAddress).settleFeeProfitAndFeeAuction(
                        tmp.collateralToken,
                        tmp.bountyFeeToProtocol,
                        0
                    );
                }
            }
        }

// (...SNIPPED...)

}
```

Listing 16.3 The *_liquidatePosition* function of the *CoreFutureClosing* contract

Recommendations

We recommend implementing a mechanism to handle scenarios where funds can potentially become permanently locked in the *APHPool* contract.

Reassessment

The *FWX* team has prevented sending the *bountyFeeToLiquidator* rewards to the *APHPool* contract as shown in the Listing 16.4.

CoreFutureClosing.sol

```
function _liquidatePosition(uint256 nftId, bytes32 pairByte) internal {
// (...SNIPPED...)
    if (msg.sender != tmp.nftOwner && poolToAsset[msg.sender] == address(0)) {
        // (...SNIPPED...)
            if (tmp.bountyFeeToLiquidator > 0) {
                _safeTransfer(tmp.collateralToken, msg.sender,
tmp.bountyFeeToLiquidator);
            }
```

Listing 16.4 The *_liquidatePosition* function of the *CoreFutureClosing* contract

No. 17	Inaccessibility Of Markets Due To Unsupported Tokens In Price Feed		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/utils/PriceFeed.sol* *contracts/src/factory/logic/FwxFactoryValidator.sol*		
Locations	*PriceFeed._queryRateUSD L: 152 -164* *FwxFactoryValidator._validateMarketCreation L: 76 - 103*		

Detailed Issue

The protocol employs the *Oracle Price Feed* to operate many operations. To elaborate, the *_queryRateUSD* function is used to query the *USD* rate of the specific token which needs the proper price feed address (L154 in the code snippet below) to perform it.

However, we found that while creating the market, it was not checked first whether there was an *Oracle Price Feed* for the trading pair they needed to create yet. This lack of verification allows users to create an unavailable market.

PriceFeed.sol

```
152  function _queryRateUSD(address token) internal view returns (uint256 rate,
     uint256 precision) {
153      require(!globalPricingPaused, "PriceFeed/pricing-is-paused");
154      require(pricesFeeds[token] != address(0), "PriceFeed/unsupported-address");
155      AggregatorV2V3Interface feed = AggregatorV2V3Interface(pricesFeeds[token]);
156      (, int256 answer, , uint256 updatedAt, ) = feed.latestRoundData();
157      rate = uint256(answer);
158      uint256 decimal = feed.decimals();
159
160      rate = (rate * WEI_PRECISION) / (10 ** decimal);
161      precision = WEI_PRECISION;
162
163      require(block.timestamp - updatedAt < stalePeriod[token],
     "PriceFeed/price-is-stale");
164  }
```

Listing 17.1 The *_queryRateUSD* function of the *PriceFeeds* contract

Recommendations

We recommend implementing the new *_validatePriceFeed* function as shown in the code snippet below. **Then apply within the *_validateMarketCreation* function of the *FwxFactoryValidator* contract to ensure the *Oracle Price Feeds* are available for operating the market**.

FwxFactoryValidator.sol

```
function _validatePriceFeed(
    address _collateralToken,
    address _underlyingTokent
) internal view returns (bool hasPriceFeeds) {
    address collateralPriceFeed =
IPriceFeed(priceFeed).pricesFeeds(_collateralTokent);
    address underlyingPriceFeed = IPriceFeed(priceFeed).
pricesFeeds(_underlyingTokent);
    if (collateral PriceFeed != address(0) &&
        underlyingPriceFeed != address(0))
    {
        hasPriceFeeds = true;
    }
}
```

Listing 17.2 The new *_validatePriceFeed* function of the *FwxFactoryValidator* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* has acknowledged this issue with the statement:

“For unsupported tokens on the oracle, the price feed will return zero. As for collateral tokens, we add the collateral token’s prerequisites of which Chainlink's Oracle price feed exists. Users who want to create markets can select only the whitelisted collateral tokens.”

No. 18	The Chainlink Oracle Rate Has The Potential To Be Either Negative Or Zero		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/utils/PriceFeed.sol* *contracts/src/core/logic/CoreSwapping.sol*		
Locations	*PriceFeed._queryRateUSD L: 152 - 164* *CoreSwapping._getAmountsWithRouterSelection L: 267 - 270*		

Detailed Issue

The *_queryRateUSD* function is designed to fetch the token price in USD, and it utilizes Chainlink's *latestRoundData* function for this purpose.

However, we've identified a potential issue. **The *latestRoundData* function provides its answer in the *int256* type.** Subsequently, the *_queryRateUSD* function **converts this answer type from *int256* to *uint256***, introducing the risk of unintended consequences.

For example, the oracle contract address could be accidentally set to an incorrect oracle contract that returns a negative integer value or zero value as the answer (L155-157 in code snippet 18.1). **The negative value will be converted to a positive integer value due to type casting,** leading to **unexpected behaviors**. In case of **a zero price**, the mechanism for checking the difference between Dex price and oracle price (L267 - 270 in code snippet 18.2) will be bypassed. **This introduces a vulnerability to potential front-running attacks within the close position flow**. The vulnerability arises because the check for the price difference with the oracle price is skipped when the fetched price is zero.

PriceFeed.sol

```
152  function _queryRateUSD(address token) internal view returns (uint256 rate, uint256 precision) {
153      require(!globalPricingPaused, "PriceFeed/pricing-is-paused");
154      require(pricesFeeds[token] != address(0), "PriceFeed/unsupported-address");
155      AggregatorV2V3Interface feed = AggregatorV2V3Interface(pricesFeeds[token]);
156      (, int256 answer, , uint256 updatedAt, ) = feed.latestRoundData();
157      rate = uint256(answer);
158      uint256 decimal = feed.decimals();
159  
160      rate = (rate * WEI_PRECISION) / (10 ** decimal);
161      precision = WEI_PRECISION;
```

```

    require(block.timestamp - updatedAt < stalePeriod[token], "PriceFeed/price-is-stale");
}
```

Listing 18.1 The *_queryRateUSD* function in *PriceFeed*

CoreSwapping.sol

```
function _getAmountsWithRouterSelection(
    bool isExactOutput,
    bytes32 pairByte,
    uint256 amountInput,
    address[] memory path,
    uint256 expectedRate,
    uint256 slippage
) internal view returns (uint256[] memory amounts, uint256 swapFee, uint256 routerIndex) {
    // (...SNIPPED...)
    rates.oracleRate = _queryOraclePrice(pairByte);
    rates.reserveRate = _getReserveRate(pairByte, routerIndex, path);
    if (
        rates.oracleRate != 0 &&
        !_checkPriceDiff(rates.oracleRate, rates.reserveRate, cfg.maxOraclePriceDiffPercent)
    ) revert("CoreSwapping/price-diff-oracle-exceed");
```

Listing 18.2 The price difference checking mechanism in *CoreSwapping* contract

Recommendations

To address this issue, **we recommend adding validation to prevent negative or zero price values** as shown below.

PriceFeed.sol

```
function _queryRateUSD(address token) internal view returns (uint256 rate, uint256 precision) {
    // (...SNIPPED...)
    AggregatorV2V3Interface feed = AggregatorV2V3Interface(pricesFeeds[token]);
    (, int256 answer, , uint256 updatedAt, ) = feed.latestRoundData();
    require(answer > 0, "PriceFeed/price-must-be-greater-than-zero");
    rate = uint256(answer);
    uint256 decimal = feed.decimals();
```

Listing 18.3 Validating the fetched price in *_queryRateUSD* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 19	Not Support Chainlink L2 Sequencer Down		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/utils/PriceFeed.sol* *contracts/src/core/logic/CoreSwapping.sol*		
Locations	*PriceFeed._queryRateUSD L: 152 - 164* *CoreSwapping._getAmountsWithRouterSelection L: 267 - 270*		

Detailed Issue

Optimistic rollup chains shift all execution away from the layer 1 (L1) Ethereum chain, completing it on a layer 2 (L2) chain, and then bringing the L2 execution results back to the L1. These protocols employ a sequencer responsible for executing and rolling up L2 transactions, grouping multiple transactions into a single transaction.

In the scenario where the protocol contracts are deployed on an **optimistic rollup-based chain**, such as Arbitrum, it is crucial to monitor the sequencer status. **If the Arbitrum Sequencer experiences downtime, the oracle data will not stay current and could become stale**. Consequently, users might interact with the protocol while oracle feeds are outdated, potentially leading to **inaccuracies in the calculation mechanism for checking the price difference between Dex price and oracle price** (L267 - 270 in code snippet 19.1).

Despite the deployment of stale price detection mechanisms (L152 - 164 in code snippet 19.2) to mitigate this issue. **There are still edge cases when the sequencer is down, but the time has not yet reached the stale period.**

CoreSwapping.sol

```
244  function _getAmountsWithRouterSelection(
245      bool isExactOutput,
246      bytes32 pairByte,
247      uint256 amountInput,
248      address[] memory path,
249      uint256 expectedRate,
250      uint256 slippage
251  ) internal view returns (uint256[] memory amounts, uint256 swapFee, uint256
     routerIndex) {
         // (...SNIPPED...)
265      rates.oracleRate = _queryOraclePrice(pairByte);
```

```
    rates.reserveRate = _getReserveRate(pairByte, routerIndex, path);
    if (
        rates.oracleRate != 0 &&
        !_checkPriceDiff(rates.oracleRate, rates.reserveRate,
cfg.maxOraclePriceDiffPercent)
    ) revert("CoreSwapping/price-diff-oracle-exceed");
```

Listing 19.1 The price difference checking mechanism in *CoreSwapping* contract

PriceFeed.sol

```
function _queryRateUSD(address token) internal view returns (uint256 rate,
uint256 precision) {
    require(!globalPricingPaused, "PriceFeed/pricing-is-paused");
    require(pricesFeeds[token] != address(0), "PriceFeed/unsupported-address");
    AggregatorV2V3Interface feed = AggregatorV2V3Interface(pricesFeeds[token]);
    (, int256 answer, , uint256 updatedAt, ) = feed.latestRoundData();
    rate = uint256(answer);
    uint256 decimal = feed.decimals();

    rate = (rate * WEI_PRECISION) / (10 ** decimal);
    precision = WEI_PRECISION;

    require(block.timestamp - updatedAt < stalePeriod[token],
"PriceFeed/price-is-stale");
}
```

Listing 19.2 The stale price detection in *PriceFeeds* contract

Recommendations

If the contracts are deployed on an optimistic rollup-based chain. We recommend adding some checks to the *PriceFeeds* contract to handle the sequencer outages as shown in the following code (https://docs.chain.link/data-feeds/l2-sequencer-feeds).

SequencerCheck.sol

```
contract SequencerCheck {
    AggregatorV2V3Interface internal sequencerUptimeFeed;

    uint256 private immutable GRACE_PERIOD_TIME;

    error GracePeriodNotOver();

    constructor(address sequencerFeedAddress, uint256 sequencerGracePeriodTime)
{
        sequencerUptimeFeed = AggregatorV2V3Interface(
            sequencerFeedAddress
        );
        GRACE_PERIOD_TIME = sequencerGracePeriodTime;
    }

    function isSequencerActive() public view returns (bool) {
        (
            /*uint80 roundID*/,
            int256 answer,
            uint256 startedAt,
            /*uint256 updatedAt*/,
            /*uint80 answeredInRound*/
        ) = sequencerUptimeFeed.latestRoundData();

        bool isSequencerUp = answer == 0;

        // Make sure the grace period has passed after the sequencer is back up.
        uint256 timeSinceUp = block.timestamp - startedAt;
        if (timeSinceUp <= GRACE_PERIOD_TIME) {
            revert GracePeriodNotOver();
        }

        return isSequencerUp;
    }
}
```

Listing 19.3 The sequencer monitoring contract

PriceFeed.sol

```
function _queryRateUSD(address token) internal view returns (uint256 rate,
uint256 precision) {
    require(!globalPricingPaused, "PriceFeed/pricing-is-paused");
    require(pricesFeeds[token] != address(0), "PriceFeed/unsupported-address");
    AggregatorV2V3Interface feed = AggregatorV2V3Interface(pricesFeeds[token]);
    (, int256 answer, , uint256 updatedAt, ) = feed.latestRoundData();
    rate = uint256(answer);
    uint256 decimal = feed.decimals();

    rate = (rate * WEI_PRECISION) / (10 ** decimal);
    precision = WEI_PRECISION;

    require(block.timestamp - updatedAt < stalePeriod[token],
"PriceFeed/price-is-stale");
    require(ISequencerCheck(sequencerCheckAddress).isSequencerActive(),
"PriceFeed/sequencer-is-down");
}
```

Listing 19.4 Monitoring sequencer status in *PriceFeeds* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team introduced the **new *PriceFeedsL2* contract**, located at *contracts/src/utils/PriceFeedL2.sol*, to specifically support price feed oracle functionality on Layer 2.

PriceFeedsL2.sol

```
function _checkUpTimeSequencer(address aggregatorAddress) internal view {
   // * if uptimeAddress exists = L2 sequencer of that aggregator exists too
   // * else no check (for L1 or non-exists sequencer chain)
   if (uptimeAddresses[aggregatorAddress] != address(0)) {
       AggregatorV2V3Interface uptimeFeed = AggregatorV2V3Interface(
           uptimeAddresses[aggregatorAddress]
       );

       (, int256 answer, uint256 startedAt, , ) = uptimeFeed.latestRoundData();
       require(answer == 0, "PriceFeed/price-sequencer-down");
       uint256 timeSinceUp = block.timestamp - startedAt;
       require(timeSinceUp > GRACE_PERIOD_TIME,
"PriceFeed/grace-period-not-over");
   }
}
```

Listing 19.5 The Layer 2 sequencer checks of the *PriceFeedsL2* contract

No. 20	Compatibility Issue With USDT Allowance Mechanism In Vault		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/utils/Vault.sol*		
Locations	*Vault._ownerApprove L: 32 - 36* *Vault.approveInterestVault L: 38 - 44*		

Detailed Issue

We found a compatibility issue with the *USDT* allowance mechanism in *Vault* on *Ethereum* chain.

Vault.sol

```
   // (...SNIPPED...)
32 function _ownerApprove(address _pool, uint256 tokenApproveAmount) internal {
33     IERC20(TOKEN).safeIncreaseAllowance(_pool, tokenApproveAmount);
34 
35     emit OwnerApproveVault(msg.sender, _pool, tokenApproveAmount);
36 }
37 
38 function approveInterestVault(
39     address _core,
40     uint256 tokenApproveAmount
41 ) external onlyAddressTimelockManager {
42     IERC20(TOKEN).safeIncreaseAllowance(_core, tokenApproveAmount);
43     emit ApproveInterestVault(msg.sender, _core, tokenApproveAmount);
44 }
   // (...SNIPPED...)
```

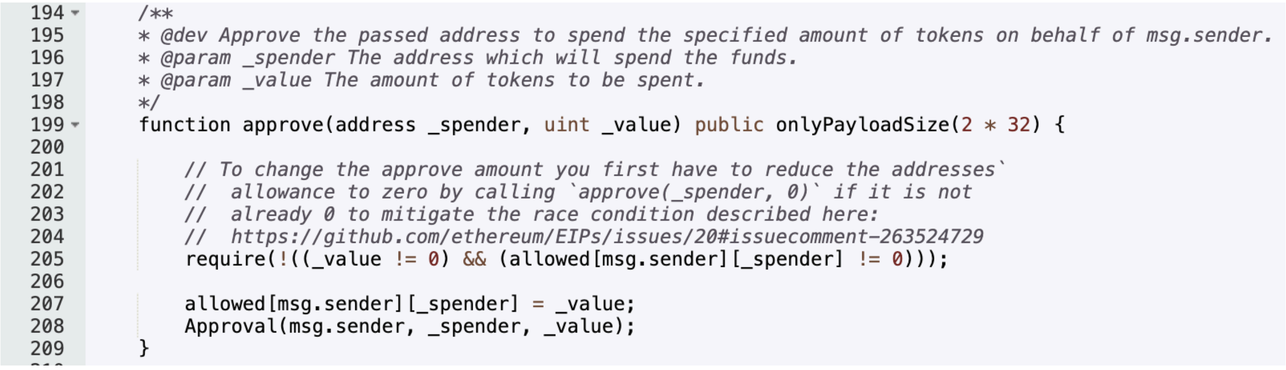
Listing 20.1 The *_ownerApprove* and *approveInterestVault* functions of the *Vault* contract

SafeERC20.sol

```
   // (...SNIPPED...)
60 function safeIncreaseAllowance(
61       IERC20 token,
62       address spender,
63       uint256 value
64 ) internal {
65       uint256 newAllowance = token.allowance(address(this), spender) + value;
66         _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
   spender, newAllowance));
67 }
```

Listing 20.2 The *safeIncreaseAllowance* function of the *SafeERC20* contract

In this case, The *safeIncreaseAllowance* function increases the allowance and calls *approve* on *USDT* (L66 in code snippet 20.2). **However, *USDT's approve* function in some blockchain networks, such as *Ethereum Mainnet*, requires the current allowance to be zero before setting a new value**, as indicated in its code at line 205. **If the entire allowance is not used, it leaves a non-zero allowance, causing subsequent non-zero approve calls to revert**. Thus, *safeIncreaseAllowance* will also revert under these conditions.



```
194     /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {
200
201         // To change the approve amount you first have to reduce the addresses`
202         //  allowance to zero by calling `approve(_spender, 0)` if it is not
203         //  already 0 to mitigate the race condition described here:
204         //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
205         require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
206
207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

The approve function in *USDT* contract on *Ethereum* chain
https://etherscan.io/address/0xdac17f958d2ee523a2206206994597c13d831ec7#code

Recommendations

We recommend to

1. First, call the *safeApprove* function with a value of 0 to reset it. Then, call safeApprove again to set the token approval amount. (both on *_ownerApprove* and *approveInterestVault* functions)

Vault.sol

```
// (...SNIPPED...)
function _ownerApprove(address _pool, uint256 tokenApproveAmount) internal {
    IERC20(TOKEN).safeApprove(_pool, 0);
    IERC20(TOKEN).safeApprove(_pool, tokenApproveAmount);
    emit OwnerApproveVault(msg.sender, _pool, tokenApproveAmount);
}
```

Listing 20.3 The improved *_ownerApprove* and *approveInterestVault* functions of the *Vault* contract

2. or use *safeIncreaseAllowance* function in contracts/token/ERC20/utils/SafeERC20.sol on **version v5.0.0** (**https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol#L52**)

Note: It's important to acknowledge that *USDT* contracts may vary across different blockchains, potentially featuring different codebases and mechanisms. Therefore, we strongly advise the team to thoroughly review and understand the specifics of the *USDT* contract on each blockchain where the *Vault* is intended to be deployed. This proactive approach will ensure compatibility and prevent any operational issues related to *USDT* allowances and interactions.

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

Vault.sol

```
// (...SNIPPED...)
function _ownerApprove(address _pool, uint256 tokenApproveAmount) internal {
    IERC20(TOKEN).safeApprove(_pool, 0);
    IERC20(TOKEN).safeApprove(_pool, tokenApproveAmount);

    emit OwnerApproveVault(msg.sender, _pool, tokenApproveAmount);
}

function approveInterestVault(
    address _core,
    uint256 tokenApproveAmount
) external onlyAddressTimelockManager {
    IERC20(TOKEN).safeApprove(_core, 0);
    IERC20(TOKEN).safeApprove(_core, tokenApproveAmount);

    emit ApproveInterestVault(msg.sender, _core, tokenApproveAmount);
}
```

Listing 20.4 The improved *_ownerApprove* and *approveInterestVault* functions of the *Vault* contract

No. 21	Missing Validating address(0) In Low-Level Delegatecall		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/factory/base/FwxFactoryProxyBase.sol*		
Locations	*FwxFactoryProxyBase._delegatecall L: 6 - 22*		

Detailed Issue

We have identified a potential issue in the *_delegateCall* function, where the lack of validation for the *address(0)* in the *targetAddress* parameter could lead to unexpected behavior.

Specifically, if targetAddress is *address(0)*, the function returns success = true, despite the call not being executed as intended.

FwxFactoryProxyBase.sol

```
 5  contract FwxFactoryProxyBase {
 6     function _delegatecall(
 7         address targetAddress,
 8         bytes memory input
 9     ) internal returns (bytes memory) {
10         // solhint-disable-next-line avoid-low-level-calls
11         (bool success, bytes memory data) = targetAddress.delegatecall(input);
12
13         if (!success) {
14             if (data.length == 0) revert("unknown-error");
15             // solhint-disable-next-line no-inline-assembly
16             assembly {
17                 revert(add(32, data), mload(data))
18             }
19         }
20         return data;
21     }
22  }
```

Listing 21.1 The *_delegateCall* function of the *FwxFactoryProxyBase* contract

Recommendations

We recommend checking *address(0)* at the beginning of the *_delegateCall* function

FwxFactoryProxyBase.sol

```
contract FwxFactoryProxyBase {
   function _delegatecall(
       address targetAddress,
       bytes memory input
   ) internal returns (bytes memory) {
       require(targetAddress != address(0), "Zero_Address targetAddress");
       // solhint-disable-next-line avoid-low-level-calls
       (bool success, bytes memory data) = targetAddress.delegatecall(input);

       if (!success) {
           if (data.length == 0) revert("unknown-error");
           // solhint-disable-next-line no-inline-assembly
           assembly {
               revert(add(32, data), mload(data))
           }
       }
       return data;
   }
}
```

Listing 21.2 The improved *_delegatecall* function of the *FwxFactoryProxyBase* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 22	Potential Inconsistency Of Crucial States		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/APHPool.sol* *contracts/src/pool/InterestVault.sol*		
Locations	*APHPool.initialize L: 20 - 67* *InterestVault.setTokenAddress L: 55 - 60* *InterestVault.setProtocolAddress L: 69 - 74*		

Detailed Issue

We have identified a potential issue concerning the inconsistency of *tokenAddress* and *protocolAddress* states within the *InterestVault* relative to the *tokenAddress* and *coreAddress* within the *APHPool*.

When the *initialize* function of *APHPool* is invoked. This function creates an *InterestVault* instance by passing *tokenAddress*, *coreAddress*, and *msg.sender* as parameters. Subsequently, the *constructor* of *InterestVault* sets its *tokenAddress* and *protocolAddress* states based on these parameters.

However, **the *tokenAddress* and *coreAddress* states of *APHPool* contract are immutable after initialization (L32, 33 in code snippet 22.1)**. Consequently, modifying **the *tokenAddress* and *protocolAddress* states in the *InterestVault* contract,** which are intently referring to those addresses**, can lead to inconsistency with the states in the *APHPool* contract.**

This inconsistency could affect functions relying on these states, leading to unexpected behaviors or vulnerabilities.

APHPool.sol

```
12 contract APHPool is PoolBaseFunc, APHPoolProxy, PoolSetting {
13     constructor() initializer {}

       /**
         @dev Function for set initial value.

         NOTE: This function must be call after deploy by deployer.
        */
20     function initialize(
21         address _logicStorage,
22         address _tokenAddress,
        address _coreAddress,
        address _membershipAddress,
        address _wethAddress,
        address _wethHandlerAddress,
        uint256 _blockTime
    ) external virtual initializer {
        require(_tokenAddress != address(0), "APHPool/initialize/tokenAddress-zero-address");
        require(_coreAddress != address(0), "APHPool/initialize/coreAddress-zero-address");
        require(_membershipAddress != address(0), "APHPool/initialize/membership-zero-address");
        tokenAddress = _tokenAddress;
        coreAddress = _coreAddress;

    // (...SNIPPED...)
    }

    // (...SNIPPED...)
}
```

Listing 22.1 The *initialize* function of the *APHPool* contract

InterestVault.sol

```
contract InterestVault is InterestVaultEvent, Ownable, SelectorPausable, ManagerTimelock {
   using SafeERC20 for IERC20;

   uint256 public claimableTokenInterest;
   uint256 public heldTokenInterest;
   uint256 public actualTokenInterestProfit;
   uint256 public cumulativeTokenInterestProfit;

   address public tokenAddress;
   address public protocolAddress;
   address public treasuryAddress;

   // (...SNIPPED...)

   function setTreasuryAddress(address _address) external onlyAddressTimelockManager {
       address oldAddress = treasuryAddress;
       treasuryAddress = _address;

       emit SetTreasuryAddress(msg.sender, oldAddress, treasuryAddress);
   }

   function setProtocolAddress(address _address) external
onlyAddressTimelockManager {
       address oldAddress = protocolAddress;
       protocolAddress = _address;

       emit SetProtocolAddress(msg.sender, oldAddress, protocolAddress);
   }

   // (...SNIPPED...)
}
```

Listing 22.2 The *crucial states and functions* of the *InterestVault* contract

Recommendations

We recommend ensuring that the *tokenAddress* and *protocolAddress* states on *InterestVault* contract must be consistent with *tokenAddress* and *coreAddress* states in *APHPool* by

1. Implementing a setter function within *APHPool* that is capable of updating the *tokenAddress* and *coreAddress*
2. **or** modifying the *tokenAddress* and *protocolAddress* states in *InterestVault* contract to be immutable

Reassessment

The *FWX* team adopted our recommended code to fix this issue by setting the *tokenAddress* and *protocolAddress* states in *InterestVault* contract to be immutable.

InterestVault.sol

```
contract InterestVault is InterestVaultEvent, Ownable, ManagerTimelock {
    using SafeERC20 for IERC20;

    // solhint-disable-next-line immutable-vars-naming
    address public immutable tokenAddress;
    // solhint-disable-next-line immutable-vars-naming
    address public immutable protocolAddress;

// (...SNIPPED...)
```

Listing 22.3 The *tokenAddress* and *protocolAddress* are immutable

No. 23	Over Deposited Amounts Are Non-Refundable		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/logic/PoolLending.sol*		
Locations	*PoolLending.depositFor L: 331 - 334*		

Detailed Issue

The *depositFor* function in the *PoolLending* contract (code snippet 23.1) is responsible for validating the deposited tokens after the caller contract executes the *depositForCallback* function (L331 - 334 in code snippet 23.1).

However, we have identified a scenario where the **caller contract deposits an excess of tokens beyond the necessary amount**. **These additional tokens become non-refundable and are locked within the contract.**

PoolLending.sol

```
295  function depositFor(
296      address caller,
297      uint256 nftId,
298      uint256 depositAmount,
299      bytes calldata data
300  )
301      external
302      payable
303      nonReentrant
304      whenFuncNotPaused(msg.sig)
305      returns (uint256 mintedP, uint256 mintedAtp, uint256 mintedItp)
306  {

         // (...SNIPPED...)

329      uint256 balanceBefore = _balance();
330      IAPHPoolCallback(msg.sender).depositForCallback(depositAmount, data);
331      require(
332          _balance() >= (balanceBefore + depositAmount),
333          "PoolLending/insufficient-input-amount"
334      );
```

```

    (mintedP, mintedAtp, mintedItp) = _deposit(caller, nftId, depositAmount);
}
```

Listing 23.1 The deposit amount validation in *PoolLending* contract

Recommendations

We recommend modifying the validation for deposited tokens, replacing the "greater than or equal to (>=)" with "*equal to (==)*," as shown in the code below.

PoolLending.sol

```
function depositFor(
    address caller,
    uint256 nftId,
    uint256 depositAmount,
    bytes calldata data
)
    external
    payable
    nonReentrant
    whenFuncNotPaused(msg.sig)
    returns (uint256 mintedP, uint256 mintedAtp, uint256 mintedItp)
{

    // (...SNIPPED...)

    uint256 balanceBefore = _balance();
    IAPHPoolCallback(msg.sender).depositForCallback(depositAmount, data);
    require(
        _balance() == (balanceBefore + depositAmount),
        "PoolLending/insufficient-input-amount"
    );

    (mintedP, mintedAtp, mintedItp) = _deposit(caller, nftId, depositAmount);
}
```

Listing 23.2 The recommended deposit amount validation in *PoolLending* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 24	Risk Of Restrictions On Future Trading Wallet		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/logic/CoreFutureWallet.sol* *contracts/src/core/logic/CoreFutureClosing.sol*		
Locations	*CoreFutureWallet.depositCollateral L: 11 - 20* *CoreFutureWallet.withdrawCollateral L: 22 - 31*		

Detailed Issue

In the *CoreFutureWallet* contract, the *depositCollateral* and *withdrawCollateral* functions can be paused. **This pause functionality can restrict users from adding or withdrawing collateral**. Specifically:

- When the ***depositCollateral*** function is paused, users are unable to add collateral. Consequently, their positions may accumulate increased borrowing interest (*interestOwed*), potentially exceeding the liquidation threshold. This can lead to undesired liquidations, despite users' readiness to bolster their collateral
- When the ***withdrawCollateral*** function is paused, users are unable to withdraw their collateral. This action effectively locks their assets within the protocol, limiting access to their funds, regardless of their intent to exit or adjust risk exposure

As a result, pausing these functions may result in unfair liquidations and inaccessible collateral.

CoreFutureWallet.sol

```
11  function depositCollateral(
12      uint256 nftId,
13      address collateralTokenAddress,
14      address underlyingTokenAddress,
15      uint256 amount
16  ) external payable nonReentrant whenFuncNotPaused(msg.sig) {
17      amount = _depositCollateral(nftId, collateralTokenAddress,
    underlyingTokenAddress, amount);
18
19      _transferFromIn(msg.sender, address(this), collateralTokenAddress, amount);
20  }
21
22  function withdrawCollateral(
23      uint256 nftId,
    address collateralTokenAddress,
    address underlyingTokenAddress,
    uint256 amount
) external nonReentrant whenFuncNotPaused(msg.sig) {
    amount = _withdrawCollateral(nftId, collateralTokenAddress,
underlyingTokenAddress, amount);

    _transferOut(msg.sender, collateralTokenAddress, amount);
}
```

Listing 24.1 The *depositCollateral* and *withdrawCollateral* functions of the *CoreFutureWallet* contract

Recommendations

We recommend that the team consider removing the pause functionality from the *depositCollateral* and *withdrawCollateral* functions, if feasible, to ensure users maintain the ability to deposit or withdraw their collateral at all times. However, this decision should be aligned with the protocol's overarching business strategy and risk management policies

Reassessment

The *FWX* team has acknowledged this issue with the statement:

"The utilization of SelectorPauseable is intended for halting functions in instances where their functionality strays from their intended purpose. According to our company policy, contract configurations can only be adjusted through a timelock contract, necessitating endorsement from a multi-signature wallet to ensure that all actions receive approval from relevant parties."

No. 25	Possibly Inconsistent Setting With The Actual Swap Fee		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/logic/CoreSwapping.sol*		
Locations	*CoreSwapping._calculateSwapFee L: 349 - 368* *CoreSwapping._getSwapFeeRate L: 441 - 454*		

Detailed Issue

The *_calculateSwapFee* function retrieves the fee rate from the *_getSwapFeeRate* function.

However, we found that the *_getSwapFeeRate* function does not get the swap fee rate from the actual decentralized exchange (DEX), instead, it is the swap rate that the admin has set at the ***swapFeeRates*** mapping.

This may be different from the actual decentralized exchange (DEX), possibly impacting the calculation that uses the price and swap fee in further calculations.

CoreSwapping.sol

```
349  function _calculateSwapFee(
350      bool isExactOutput,
351      uint256 routerIndex,
352      uint256[] memory amounts,
353      address[] memory path
354  ) internal view returns (uint256[] memory resultAmounts, uint256 swapFee) {
355      resultAmounts = amounts;
356      uint256 swapFeeRate = _getSwapFeeRate(routerIndex, path[0], path[1]);
357
358      if (isExactOutput) {
359          swapFee = (amounts[0] * swapFeeRate) / WEI_PERCENT_UNIT;
360          resultAmounts[0] -= swapFee;
361      } else {
362          (uint256 reserve0, ) = _getReserves(routerIndex, path[0], path[1]);
363          swapFee =
364              (reserve0 * amounts[1] * swapFeeRate) /
365              ((reserve0 + amounts[0]) * (WEI_PERCENT_UNIT - swapFeeRate));
366          resultAmounts[1] += swapFee;
367      }
368  }
```

Listing 25.1 The *_calculateSwapFee* function of the *CoreSwapping* contract

CoreSwapping.sol

```
function _getSwapFeeRate(
    uint256 routerIndex,
    address token0,
    address token1
) internal view returns (uint256 swapFeeRate) {
    if (routerIndex == 0) {
        // Other router's swap fee rate
        swapFeeRate = swapFeeRates[routers[routerIndex]];
    } else {
        // disable solc warning
        token0;
        token1;
    }
}
```

Listing 25.2 The *_getSwapFeeRate* function of the *CoreSwapping* contract

Recommendations

There is no recommendation code for this issues as it might break the contract functionality and require a decision from the *FWX* team in terms of business and protocol's core functionality,

However, we recommend the *FWX* team ensure the swap fee setting is consistent with the actual decentralized exchange (*DEX*).

Reassessment

The *FWX* team has acknowledged this issue with the statement:

"Since the swap fee rate of external routers (DEXs) cannot be derived programmatically from the smart contracts, we have to configure them manually. However, the development and research teams must ensure that configurations in the smart contracts are set from a multi-signature wallet properly."

No. 26	Potentially Underflow Revert On Bounty Fee Distribution		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/logic/CoreFutureClosing.sol*		
Locations	*CoreFutureClosing._liquidatePosition L: 106 - 230*		

Detailed Issue

The *tmp.bountyFeeToProtocol* and *tmp.bountyFeeToLiquidator* variables are separately calculated based on the *bountyFeeRateToProtocol* and *bountyFeeToLiquidator* configurations and then subtracted from the user's wallet.

However, there is no handling for the case of ***bountyFeeRateToProtocol* + *bountyFeeToLiquidator* > *100%***, resulting in an execution revert due to arithmetic underflow.

CoreFutureClosing.sol

```
106  function _liquidatePosition(uint256 nftId, bytes32 pairByte) internal {

         // (...SNIPPED...)

159      if (msg.sender != IMembership(membershipAddress).ownerOf(nftId)) {
160          // bounty fee
161          {
162              uint256 wallet = wallets[nftId][pairByte];
163
164              (uint256 rate, ) = _queryRateUSD(tmp.collateralToken);
165              uint256 collateralPrecision =
     tokenPrecisionUnit[tmp.collateralToken];
166              uint256 feeToLiquidator = (liquidationFee * collateralPrecision) /
     rate;
167
168              if (feeToLiquidator >= wallet) {
169                  feeToLiquidator = wallet;
170                  wallet = 0;
171              } else {
172                  wallet = wallet - feeToLiquidator;
173
174                  tmp.bountyFeeToProtocol =
                    (wallet * positionConfigs[pairByte].bountyFeeRateToProtocol)
/
                    WEI_PERCENT_UNIT;
                tmp.bountyFeeToLiquidator =
                    (wallet *
positionConfigs[pairByte].bountyFeeRateToLiquidator) /
                    WEI_PERCENT_UNIT;

                wallet = wallet - tmp.bountyFeeToProtocol -
tmp.bountyFeeToLiquidator;
            }

    // (...SNIPPED...)
```

Listing 26.1 The *_liquidatePosition* function of the *CoreFutureClosing* contract

Recommendations

We recommend implementing a boundary check to handle scenarios where *tmp.bountyFeeToProtocol* and *tmp.bountyFeeToLiquidator* can be subtracted from the *wallet* variable without triggering underflow reverts.

Moreover, we recommend considering the case that the accumulation of *bountyFeeRateToProtocol* + *bountyFeeRateToProtocol* configurations exceeds 100%.

Reassessment

The *FWX* team fixed this issue by adding the boundary check before setting the ***bountyFeeRateToProtocol*** and ***bountyFeeRateToLiquidator*** as shown in the code snippet below.

CoreSetting.sol

```
function setPositionConfig(
    address collateralTokenAddress,
    address underlyingTokenAddress,
    uint256 maintenanceMargin,
    uint256 minimumMargin,
    uint256 bountyFeeRateToProtocol,
    uint256 bountyFeeRateToLiquidator,
    uint256 minOpenPositionSize,
    uint256 maxOpenPositionSize
) external onlyConfigTimelockManager {
    require(
        bountyFeeRateToProtocol + bountyFeeRateToLiquidator <= WEI_PERCENT_UNIT,
        "CoreSetting/invalid-bounty-fee"
    );
```

```
// (...SNIPPED...)
```

Listing 26.2 The improved *setPositionConfig* function of the *CoreSetting* contract

FwxFactorySetting.sol

```
function setPositionConfig(
    uint256 _maintenanceMargin,
    uint256 _minimumMargin,
    uint256 _bountyFeeRateToProtocol,
    uint256 _bountyFeeRateToLiquidator,
    uint256 _forwRewardAmount,
    uint256 _positionSizeTargetInUSD,
    uint256 _minOpenPositionSize,
    uint256 _maxOpenPositionSize
) external onlyConfigTimelockManager {
    require(
        _bountyFeeRateToProtocol + _bountyFeeRateToLiquidator <= WEI_PERCENT_UNIT,
        "CoreSetting/invalid-bounty-fee"
    );

// (...SNIPPED...)
```

Listing 26.3 The improved *setPositionConfig* function of the *FwxFactorySetting* contract

No. 27	Potentially Underflow Revert On Profit Distribution		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/logic/PoolLending.sol*		
Locations	*PoolLending._claimTokenInterest L: 252 - 293*		

Detailed Issue

We found that the ***bonusAmount*** could possibly be greater than the left side of the ***profiAmount*** calculation L279. This can cause transactions to always revert by underflow reverts, preventing the execution of functions that apply the *_claimTokenInterest* function.

PoolLending.sol

```
252 function _claimTokenInterest(
253         address receiver,
254         uint256 nftId,
255         uint256 claimAmount
256     ) internal returns (WithdrawResult memory result) {
257         uint256 itpPrice = _getInterestTokenPrice();
258         PoolTokens storage tokenHolder = tokenHolders[nftId];
259 
260         uint256 claimableAmount;
261         if (((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) >
    tokenHolder.pToken) {
262             claimableAmount =
263                 ((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) -
264                 tokenHolder.pToken;
275         }
266 
267         claimAmount = MathUpgradeable.min(claimAmount, claimableAmount);
268 
269         uint256 burnAmount = _burnItpToken(
270             receiver,
271             nftId,
272             (claimAmount * PRECISION_UNIT) / itpPrice,
273             itpPrice
274         );
275         uint256 bonusAmount = (claimAmount *
    _getPoolRankInfo(nftId).interestBonusLending) /
```

```
            WEI_PERCENT_UNIT;

        uint256 feeSpread = IAPHCore(coreAddress).feeSpread();
        uint256 profitAmount = ((claimAmount * feeSpread) / (WEI_PERCENT_UNIT -
feeSpread)) - bonusAmount;

// (...SNIPPED...)

}
```

Listing 27.1 The *_claimTokenInterest* function of the *PoolLending* contract

Recommendations

We recommend implementing a boundary check to handle the subtraction of *bonusAmount* without triggering arithmetic underflow reverts.

Reassessment

The *FWX* team applied the boundary check of the *bonusAmount* against the *profitAmount* to prevent the underflow revert.

PoolLending.sol

```
function _claimTokenInterest(
        address receiver,
        uint256 nftId,
        uint256 claimAmount
) internal returns (WithdrawResult memory result) {

// (...SNIPPED...)

    uint256 bonusAmount = (claimAmount *
_getPoolRankInfo(nftId).interestBonusLending) /
    WEI_PERCENT_UNIT;


    uint256 feeSpread = IAPHCore(coreAddress).feeSpread();
    uint256 profitAmount = ((claimAmount * feeSpread) / (WEI_PERCENT_UNIT -
feeSpread));
    profitAmount -= MathUpgradeable.min(bonusAmount, profitAmount);

// (...SNIPPED...)

}
```

Listing 27.2 The improved *_claimTokenInterest* function of the *PoolLending* contract

No. 28	Potentially Underflow Revert On The withdrawTokenInterest Function		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/InterestVault.sol*		
Locations	*InterestVault.withdrawTokenInterest L: 101 - 107* *InterestVault._withdrawTokenInterest L: 135 - 142*		

Detailed Issue

We discovered a potential issue where transactions may revert in the *_withdrawTokenInterest* function if the *claimable* parameter exceeds the *claimableTokenInterest* state (L136 in code snippet below). This can cause transactions to always revert, preventing crucial state updates.

InterestVault.sol

```
135  function _withdrawTokenInterest(uint256 claimable, uint256 bonus, uint256
     profit) internal {
136      claimableTokenInterest -= claimable;
137      heldTokenInterest -= bonus + profit;
138      actualTokenInterestProfit += profit;
139      cumulativeTokenInterestProfit += profit;
140
141      emit WithdrawTokenInterest(msg.sender, claimable, bonus, profit);
142  }
```

Listing 28.1 The *_withdrawTokenInterest* function of the *InterestVault* contract

Recommendations

We recommend

1. Ensure *claimable* parameter must be less than or equal to *claimableTokenInterest* state
2. Ensure the sum of *bonus* and *profit* parameters must be less than or equal to *heldTokenInterest* state
3. Both steps may use the *min* function in Math library to check the minimum value before subtraction
   import {Math} from *"@openzeppelin/contracts/utils/math/Math.sol";*

Reassessment

The *FWX* team has prevented the underflow revert on the *withdrawTokenInterest* function by ensuring that

1. The *claimedInterest* must be less than or equal to *claimableTokenInterest* state.
2. The sum of *claimedBonus* and *claimedProfit* must be less than or equal to *heldTokenInterest* state.

The fix is shown in the Listing 28.2.

InterestVault.sol

```
function _withdrawTokenInterest(
    uint256 claimable,
    uint256 bonus,
    uint256 profit
) internal returns (uint256 claimedInterest, uint256 claimedBonus, uint256 claimedProfit) {
    claimedInterest = Math.min(claimable, claimableTokenInterest);
    if (bonus > heldTokenInterest) {
        claimedBonus = heldTokenInterest;
        claimedProfit = 0;
    } else if (bonus + profit > heldTokenInterest) {
        claimedBonus = bonus;
        claimedProfit = heldTokenInterest - bonus;
    } else {
        claimedBonus = bonus;
        claimedProfit = profit;
    }

    claimableTokenInterest -= claimedInterest;
    heldTokenInterest -= claimedBonus + claimedProfit;
    actualTokenInterestProfit += profit;
    cumulativeTokenInterestProfit += profit;

    emit WithdrawTokenInterest(msg.sender, claimedInterest, claimedBonus, claimedProfit);
}
```

Listing 28.2 The improved _*withdrawTokenInterest* function of the *InterestVault* contract

No. 29	**Lack Of Support For Multiple Routers Configuration**		
Risk	**Medium**	**Likelihood**	**Medium**
		Impact	**Medium**
Functionality is in use	**In use**	**Status**	**Acknowledged**
Associated Files	*contracts/src/core/logic/CoreSwapping.sol*		
Locations	*CoreSwapping.loanLiquidationSwap L: 60 - 77* *CoreSwapping.loanLiquidationSwap L: 93 - 123* *CoreSwapping._getAmountsWithRouterSelection L: 244 - 273*		

Detailed Issue

The automatic future trading hedging system uses the *Decentralized Exchange (DEX)* to swap the target token to hedge the user's trading in the protocol, therefore, the swap price relies on the amount of swapping and *DEX* liquidity.

We notice the protocol is designed to support multiple routers as shown in the code snippet below.

CoreBase.sol

```
contract CoreBase is
    AssetHandlerUpgradeable,
    ManagerTimelockUpgradeable,
    ReentrancyGuardUpgradeable,
    SelectorPausableUpgradeable
{

    // (...SNIPPED...)

    address[5] public routers; //
// list of routers addresses (max: 5)
    mapping(address => mapping(bytes32 => SwapConfig)) public swapConfigs; //
// router => pairByte => router limit
    mapping(address => uint256) public swapFeeRates; //
// router => swap fee rate of the router
    uint256 public forwStakingMultiplier;

    // solhint-disable-next-line var-name-mixedcase
    uint256[50] private __gap_bottom_coreBase;
}
```

Listing 29.1 The *routers* state of the *CoreBase* contract

However, the current implementation does not support multiple routers, by always using the router index 0 to perform swapping as shown in the code snippets below.

This lack of ability to select the router for swapping may create the risk of price impact that affects the user trading.

CoreSwapping.sol

```
function _getAmountsWithRouterSelection(
    bool isExactOutput,
    bytes32 pairByte,
    uint256 amountInput,
    address[] memory path,
    uint256 expectedRate,
    uint256 slippage
) internal view returns (uint256[] memory amounts, uint256 swapFee, uint256 routerIndex) {
    routerIndex = 0;
    Rates memory rates;
    SwapConfig memory cfg = swapConfigs[routers[routerIndex]][pairByte];

    // verifying for external dex
    if (!_isRouterUsable(routerIndex, isExactOutput, amountInput, path))
        revert("CoreSwapping/cannot-find-usable-router");

    (amounts, swapFee) = _getAmounts(isExactOutput, true, routerIndex, amountInput, path);
    rates.swapRate = _calculateSwapRate(pairByte, path, amounts);
    if (slippage != 0 && !_checkPriceDiff(expectedRate, rates.swapRate, slippage))
        revert("CoreSwapping/slippage-too-low");

    rates.oracleRate = _queryOraclePrice(pairByte);
    rates.reserveRate = _getReserveRate(pairByte, routerIndex, path);
    if (
        rates.oracleRate != 0 &&
        !_checkPriceDiff(rates.oracleRate, rates.reserveRate, cfg.maxOraclePriceDiffPercent)
    ) revert("CoreSwapping/price-diff-oracle-exceed");

    return (amounts, swapFee, routerIndex);
}
```

Listing 29.2 The *_getAmountsWithRouterSelection* function of the *CoreSwapping* contract

CoreSwapping.sol

```solidity
function loanLiquidationSwap(
    bool isExactOutput,
    uint256 amountIn,
    uint256 amountOut,
    address[] memory path,
    address receiver
) external returns (uint256[] memory amounts, uint256 swapFee, address router) {
    uint256 routerIndex = 0; // external dex
    router = routers[routerIndex];
    (amounts, swapFee) = _getAmounts(
        isExactOutput,
        true,
        routerIndex,
        isExactOutput ? amountOut : amountIn,
        path
    );
    _swap(isExactOutput, routerIndex, amountIn, amountOut, path, receiver);
}
```

Listing 29.3 The *loanLiquidationSwap* function of the *CoreSwapping* contract

CoreSwapping.sol

```solidity
function positionLiquidationSwap(
    bool isExactOutput,
    bytes32 pairByte,
    uint256 amountIn,
    uint256 amountOut,
    address[] memory path,
    address receiver
) external returns (uint256[] memory amounts, uint256 swapFee, address router) {
    uint256 routerIndex = 0; // external dex
    uint256 oracleRate = _queryOraclePrice(pairByte);

    // get actual rate from external dex
    router = routers[routerIndex];
    (amounts, swapFee) = _getAmounts(
        isExactOutput,
        true,
        routerIndex,
        isExactOutput ? amountOut : amountIn,
        path
    );
    uint256 swapRate = _calculateSwapRate(pairByte, path, amounts);

    // compare actual rate to oracle rate
    SwapConfig memory cfg = swapConfigs[router][pairByte];
    require(
        oracleRate == 0 ||
            _checkPriceDiff(oracleRate, swapRate, cfg.maxLiquidationOraclePriceDiffPercent),
        "CoreSwapping/liquidate-price-diff-oracle-exceed"
    );
    _swap(isExactOutput, routerIndex, amountIn, amountOut, path, receiver);
}
```

Listing 29.4 The *positionLiquidationSwap* function of the *CoreSwapping* contract

Recommendations

There is no recommendation code for this issue as it might break the contract functionality and requires the decision from the *FWX* team in terms of business and protocol's core functionality.

However, we recommend re-implementing the swap mechanism to support multiple routers to mitigate the risk of price impact from the single *Decentralized Exchange (DEX)*.

Reassessment

The *FWX* team has acknowledged this issue with the statement:

"In this version of Permissionless, markets are unified under a single DEX router, enhancing liquidity and accessibility. Bringing markets together simplifies operations and makes the platform more robust."

No. 30	Inconsistency In Fee, Trading Fee And Auction Spread Validation		
Risk	Medium	Likelihood	Medium
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/logic/CoreSetting.sol*		
Locations	*CoreSetting.setFeeSpread L: 64 - 70* *CoreSetting.setTradingFeeToLender L: 72 - 77* *CoreSetting.setAuctionSpread L: 79 - 84*		

Detailed Issue

The functions responsible for managing fees include *setFeeSpread*, *setTradingFeeToLender* and *setAuctionSpread*. These functions are implemented with a mechanism to validate the maximum fee.

We've identified inconsistencies in this validation. **While *feeSpread* can be set up to 100 percent, *tradingFeeToLender* and *auctionSpread* can be set up to only 99 percent.** Additionally, the error messages for invalid *tradingFeeToLender* and *auctionSpread* are ***Value_Exceed_100_Percent*** (L73 and L80 in the code snippet below), which is incorrect because the **value can not actually be set to 100 percent**.

However, the team should verify the maximum fee and the error messages to be aligned with the protocol's business strategy.

CoreSetting.sol

```
64  function setFeeSpread(uint256 _value) external onlyConfigTimelockManager {
65      require(_value <= WEI_PERCENT_UNIT, "CoreSetting/value-exceed-100-percent");
66      uint256 oldValue = feeSpread;
67      feeSpread = _value;
68
69      emit SetFeeSpread(msg.sender, oldValue, _value);
70  }
71
72  function setTradingFeeToLender(uint256 _value) external
    onlyConfigTimelockManager {
73      require(_value < WEI_PERCENT_UNIT, "Value_Exceed_100_Percent");
74      uint256 oldValue = tradingFeeToLender;
75      tradingFeeToLender = _value;
76      emit SetTradingFeeToLender(msg.sender, oldValue, _value);
77  }
```

```

function setAuctionSpread(uint256 _value) external onlyConfigTimelockManager {
    require(_value < WEI_PERCENT_UNIT, "Value_Exceed_100_Percent");
    uint256 oldValue = auctionSpread;
    auctionSpread = _value;
    emit SetAuctionSpread(msg.sender, oldValue, _value);
}
```

Listing 30.1 The *setTradingFeeToLender* and *setAuctionSpread* functions of the *CoreSetting* contract

Recommendations

We recommend adjusting the maximum fee or the error messages, so that they are consistent with each other. However, it is important for the team to review the modified code to ensure it aligns with the protocol's business strategy.

Reassessment

The *FWX* team adopted our recommended code to fix this issue and improve the revert message.

CoreSetting.sol

```
function setTradingFeeToLender(uint256 _value) external
onlyConfigTimelockManager {
    require(_value <= WEI_PERCENT_UNIT, "CoreSetting/fee-to-lender-exceed-limit");
    uint256 oldValue = tradingFeeToLender;
    tradingFeeToLender = _value;
    emit SetTradingFeeToLender(msg.sender, oldValue, _value);
}

function setAuctionSpread(uint256 _value) external onlyConfigTimelockManager {
    require(_value <= WEI_PERCENT_UNIT, "CoreSetting/auction-spread-exceed-limit");
    uint256 oldValue = auctionSpread;
    auctionSpread = _value;
    emit SetAuctionSpread(msg.sender, oldValue, _value)
```

Listing 30.2 The improved *setTradingFeeToLender* and *setAuctionSpread* functions of the *CoreSetting* contract

No. 31	Improperly Getting Total Token Interest		
Risk	Medium	Likelihood	High
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/InterestVault.sol*		
Locations	*InterestVault.getTotalTokenInterest L: 116 - 118*		

Detailed Issue

We found a potential mismatch in the *getTotalTokenInterest* function of the *InterestVault* contract, which returns the *ERC20* token balance of the interest vault.

This value might not accurately reflect the actual token interest profits tracked by the contract (i.e., *actualTokenInterestProfit*). **Upon examination, we found a potential mismatch that could occur if additional tokens are directly sent into the *InterestVault* by an attacker or a grifter. This action increases the balance and could potentially deceive external protocols or off-chain mechanisms that rely on this function for accurate interest information.**

InterestVault.sol

```
116  function getTotalTokenInterest() external view returns (uint256) {
117      return IERC20(tokenAddress).balanceOf(address(this));
118  }
```

Listing 31.1 The *getTotalTokenInterest* function of the *InterestVault* contract

Recommendations

As there are several factions of interest tracked in the *InterstVault* contract, we recommend the *FWX* team to ensure the actual intent of the *getTotalTokenInterest* function behavior.

Alternatively, renaming the *getTotalTokenInterest* function to *getTotalInterestBalanceOfInterestVault* or a similar name that aligns with the behavior of *ERC20.balanceOf(address(this))* could also clarify the function's purpose and prevent misunderstandings about the returned value.

Reassessment

The *FWX* team updates the *getTotalTokenInterest* function to return the accumulated value of the interest.

InterestVault.sol

```
// (...SNIPPED...)

function getTotalTokenInterest() external view returns (uint256) {
    return claimableTokenInterest + heldTokenInterest;
}
```

Listing 31.1 The improved *getTotalTokenInterest* function of the *InterestVault* contract

No. 32	Incorrect Behavior Of Usable NFT		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/logic/PoolLending.sol*		
Locations	*PoolLending.depositFor L: 295 - 337*		

Detailed Issue

We found that the *depositFor* function accepts any address as a parameter for the *caller* address, including smart contract addresses. **This function does not implement a check to verify whether the NFT ID owner is an External Owned Account (EOA) or a smart contract.**

This **overlooks the platform's intended design, restricting NFT ownership to only EOAs**. It could potentially lead to unintended interactions and complications, given the platform's primary design for EOA interactions.

PoolLending.sol

```
295 function depositFor(
296     address caller,
297     uint256 nftId,
298     uint256 depositAmount,
299     bytes calldata data
300 )
301     external
302     payable
303     nonReentrant
304     whenFuncNotPaused(msg.sig)
305     returns (uint256 mintedP, uint256 mintedAtp, uint256 mintedItp)
306 {
307     /**
308      * NOTE
309      *   caller      = user
310      *   msg.sender  = FwxFactory
311      */
312 
313     require(msg.value == 0, "PoolLending/unsupported-native-token");
314     require(
315         caller == IMembership(membershipAddress).ownerOf(nftId),
        "PoolLending/deposit-for-unowned-nft"
    );

    // (...SNIPPED...)
}
```

Listing 32.1 The *depositFor* function of the *PoolLending* contract

Recommendations

There is no recommendation code for this issue as it might break the contract functionality and requires the decision from the *FWX* team in terms of business and protocol's core functionality.

Reassessment

The *FWX* team implemented a flag (***isFactoryInitiated***) that ensures the factory is invoked only once during the market creation flow. This eliminates the potential issue mentioned earlier.

PoolLending.sol

```
function depositFor(
    address caller,
    uint256 nftId,
    uint256 depositAmount,
    bytes calldata data
)
    external
    nonReentrant
    whenFuncNotPaused(msg.sig)
    onlyNoTimelockManager
    returns (uint256 mintedP, uint256 mintedAtp, uint256 mintedItp)
{
    /**
        * NOTE
        *  caller      = user
        *  msg.sender  = FwxFactory
        *
        *  When the pool is deployed, the FwxFactory will be all noTimelockManager, configTimelockManager, and addressTimelockManager.
        *  After this function is called, the managers will be transferred from FwxFactory to multi-signature accounts.
        */
    require(!isFactoryInitiated, "PoolLending/depositFor-disabled");

     // (...SNIPPED...)
}
```

Listing 32.2 The *depositFor* function of the *PoolLending* contract

No. 33	Incorrect collateralSwappedAmount Return Event Emission Value		
Risk	Medium	Likelihood	High
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/logic/CoreFutureClosing.sol*		
Locations	*CoreFutureClosing._closeLong L: 232 - 343*		

Detailed Issue

We notice that the end of the *_closePosition* function needs to emit the *collateralSwappedAmountReturn* result to the *ClosePosition* event (L91 in the code snippet below).

However, we found that the *_closeLong* function does not return the calculated *collateralSwappedAmountReturn* (L295 - 298 in code snippet 33.2) to properly emit the event, affecting the transparency and traceability of the protocol.

CoreFutureClosing.sol

```
24  function _closePosition(uint256 nftId, uint256 _posId, uint256 _closingSize)
    internal {

            // (...SNIPPED...)

            // close position
65          result = posState.isLong ? _closeLong(params) : _closeShort(params);
66
67          // transfer fee to interest vault and profit vault
68          _settleAndTransferFutureTradeFee(
69              posState.isLong ? pair.pair0 : pair.pair1,
70              result.feeToIntVault,
71              result.feeToProfitVault
72          );
73
74          // repay borrowing tokens back to pool.
75          _safeTransfer(
76              posState.isLong ? pair.pair0 : pair.pair1,
77              assetToPool[posState.isLong ? pair.pair0 : pair.pair1],
78              result.repayAmount
79          );
80
```

```
        emit ClosePosition(
            msg.sender,
            nftId,
            _posId,
            params.closingSize,
            result.rate,
            result.pnl,
            posState.isLong,
            !posState.active,
            posState.pairByte,
            result.collateralSwappedAmountReturn,
            result.router,
            uint128(result.tradingFee),
            uint128(result.swapFee)
        );

        // (...SNIPPED...)
```

Listing 33.1 The *_closePosition* function of the *CoreFutureClosing* contract

CoreFutureClosing.sol

```
function _closeLong(
    APHLibrary.ClosePositionParams memory params
) internal returns (APHLibrary.ClosePositionResponse memory result) {

    // (...SNIPPED...)

        _updateWallet(params.nftId, params.pairByte, actualCollateral);

        uint256 newInterestOwedPerDay = (pos.interestOwePerDay *
            (pos.contractSize - params.closingSize)) / pos.contractSize;
        uint256 collateralSwappedAmountReturn = MathUpgradeable.min(
            (pos.collateralSwappedAmount * params.closingSize) /
pos.contractSize,
            pos.collateralSwappedAmount
        );

    // (...SNIPPED...)

}
```

Listing 33.2 The *_closeLong* function of the *CoreFutureClosing* contract

Recommendations

We recommend re-implementing the *_closeLong* function to return the *collateralSwappedAmountReturn* parameter that is used to emit the event for transparency and traceability of the protocol.

Reassessment

The *FWX* team has acknowledged this issue with the statement:

"It works as design. We need to emit collateralSwappedAmountReturn as the amount of collateral released from locked.

For long: we swap collateral when the position opened and swap back when the position closed so that it doesn't return locked collateral.

For short: we lock collateral when the position is opened and it is released after the position closes."

From the statement, the status of this issue can be marked as ***Fixed*** as it functions as designed.

No. 34	Recommended Following Best Practices For Upgradeable Smart Contracts		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/factory/logic/FwxFactoryLogic.sol* *contracts/src/factory/logic/FwxFactorySetting.sol* *contracts/src/factory/logic/FwxFactoryValidator.sol* *contracts/src/pool/APHPool.sol* *contracts/src/core/APHCore.sol*		
Locations	*Several constructor of associated files*		

Detailed Issue

The following contracts should enhance the disable initializer mechanism to be broadly supported in future upgrades and follow the best practices.

- The *FwxFactoryLogic* contract
- The *FwxFactorySetting* contract
- The *FwxFactoryValidator* contract
- The *APHPool* contract
- The *APHCore* contract

APHCore.sol

```
contract APHCore is APHCoreProxy, APHCoreSettingProxy, CoreEvent, CoreSettingEvent {
    constructor() initializer {}

    // (...SNIPPED...)
```

Listing 34.1 The example disable initializer mechanism which does not protect in the case of the contract upgrades

The practice above performs equivalent to ***reinitializer(1)*** which does not protect in the case of the contract upgrades that require reinitialization of the next version (version > 1).

Recommendations

We recommend revising to use the ***_disableInitializers*** function.

The ***_disableInitializers*** function **guards against future reinitializations** by setting *_initialized* version to the max supported version (uint8.max for *OpenZeppelin* contract version <= v4.9.5, uint64.max, >= v5.0.0, for *OpenZeppelin* contract version).

APHCore.sol

```
contract APHCore is APHCoreProxy, APHCoreSettingProxy, CoreEvent,
CoreSettingEvent {
    constructor() {
        _disableInitializers();
    }

    // (...SNIPPED...)
```

Listing 34.1 The example revising to use the *_disableInitializers* function

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 35	Unsafe ABI Encoding		
Risk	Low	Likelihood	Low
		Impact	Medium
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/logic/CoreBaseFunc.sol* *contracts/src/core/proxy/APHCoreProxy.sol* *contracts/src/core/proxy/APHCoreSettingProxy.sol* *contracts/src/pool/APHPoolProxy.sol* *contracts/src/factory/logic/FwxFactoryLogic.sol* *contracts/src/factory/proxy/FwxFactoryLogicProxy.sol* *contracts/src/factory/proxy/FwxFactorySettingProxy.sol* *contracts/src/factory/proxy/FwxFactoryValidatorProxy.sol* *contracts/src/pool/logic/PoolLending.sol*		
Locations	*Several functions throughout multiple contracts*		

Detailed Issue

We found that the use of ***abi.encodeWithSignature*** and ***abi.encodeWithSelector*** functions for generating *calldata* in low-level calls introduce potential risks in several functions.

The first function is susceptible to typographical errors, and the second lacks type safety. These vulnerabilities can lead to unexpected and unsafe outcomes in smart contract operations.

CoreBaseFunc.sol

```
function _swap(
    bool isExactOutput,
    bytes32 pairByte,
    uint256 amountIn,
    uint256 amountOut,
    address src,
    address dst,
    address receiver,
    uint256 expectedRate,
    uint256 slippage
) internal returns (uint256[] memory amounts, uint256 swapFee, address router) {
    bytes memory data = abi.encodeWithSignature(
        "swap(bool,bytes32,uint256,uint256,address[],address,uint256,uint256)",
        isExactOutput,
        pairByte,
        amountIn,
        amountOut,
        _createPath(src, dst),
        receiver,
        expectedRate,
        slippage
    );

    data = _delegateCall(ILogicStorage(logicStorageAddress).coreSwappingAddress(), data);
    (amounts, swapFee, router) = abi.decode(data, (uint256[], uint256, address));
}
```

Listing 35.1 The *_swap* function of the *CoreBaseFunc* contract

FwxFactoryLogicProxy.sol

```
function createMarket(
    uint256 nftId,
    address collateralToken,
    address underlyingToken,
    uint256 collateralTokenSent,
    uint256 underlyingTokenSent
) external override returns (address core, address collateralPool, address underlyingPool) {
    bytes memory data = abi.encodeWithSelector(
        IFwxFactoryLogic.createMarket.selector,
        nftId,
        collateralToken,
        underlyingToken,
        collateralTokenSent,
        underlyingTokenSent
    );
    data = _delegatecall(fwxFactory, data);
    return abi.decode(data, (address, address, address));
}
```

Listing 35.2 The *createMarket* function of the *FwxFactoryLogicProxy* contract

Recommendations

We recommend replacing any instances of unsafe ABI encodings with ***abi.encodeCall***, which verifies that the given values match the types anticipated by the called function while avoiding typographical errors.

Reference from [docs.soliditylang.org]()
abi.encodeCall(function functionPointer, (...)) returns (bytes memory)**:** ABI-encodes a call to functionPointer with the arguments found in the tuple. Performs a full type-check, ensuring the types match the function signature.

Reassessment

The *FWX* team adopted our recommended code to fix this issue by updating to use the ***abi.encodeCall*** for encoding.

No. 36	Incomplete Legacy Data Removal In Utils Rates		
Risk	Low	Likelihood	Medium
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/logic/PoolSetting.sol*		
Locations	*PoolSetting.setBorrowInterestParams L: 9 - 32*		

Detailed Issue

The *setBorrowInterestParams* function serves as a setter for the interest calculation of each borrow.

However, We have identified inconsistencies in the process. **Although the function updates the *rates* and *utils* states with new values, it does not remove the old values completely(L24 - 27 in the *PoolSetting* contract).** Let's consider, the following scenario

1. The current *rates* are [50, 60, 70] and the current *utils* are [0, 10, 55].
2. The setBorrowInterestParams function is invoked with [80, 90] as the new *rates* and [50, 60] as the new *utils.* Now, the rates are set as [80, 90, 70] and the utils are set as [50, 60, 55].

However, some old values are still left in the states. This inconsistency may impact the traceability of the protocol.

PoolSetting.sol

```
 9  function setBorrowInterestParams(
10       uint256[] memory _rates,
11       uint256[] memory _utils,
12       uint256 _targetSupply
13   ) external onlyConfigTimelockManager {
14       require(_rates.length == _utils.length, "PoolSetting/length-not-equal");
15       require(_rates.length <= 10, "PoolSetting/length-too-high");
16       require(_utils[0] == 0, "PoolSetting/invalid-first-util");
17       require(_utils[_utils.length - 1] == WEI_PERCENT_UNIT,
    "PoolSetting/invalid-last-util");
18
19       for (uint256 i = 1; i < _rates.length; i++) {
20           require(_rates[i - 1] <= _rates[i], "PoolSetting/invalid-rate");
21           require(_utils[i - 1] < _utils[i], "PoolSetting/invalid-util");
22       }
23
```

```
    for (uint256 i = 0; i < _rates.length; i++) {
        rates[i] = _rates[i];
        utils[i] = _utils[i];
    }
    targetSupply = _targetSupply;
    utilsLen = _utils.length;

    emit SetBorrowInterestParams(msg.sender, _rates, _utils, targetSupply);
}
```

Listing 36.1 The *setBorrowInterestParams* function of the *PoolSetting* contract

Recommendations

We recommend resetting the *rates* and *utils* states to be empty before setting them to new values (as shown in the code snippet below).

PoolSetting.sol

```
function setBorrowInterestParams(
    uint256[] memory _rates,
    uint256[] memory _utils,
    uint256 _targetSupply
 ) external onlyConfigTimelockManager {
    require(_rates.length == _utils.length, "PoolSetting/length-not-equal");
    require(_rates.length <= 10, "PoolSetting/length-too-high");
    require(_utils[0] == 0, "PoolSetting/invalid-first-util");
    require(_utils[_utils.length - 1] == WEI_PERCENT_UNIT,
"PoolSetting/invalid-last-util");

    for (uint256 i = 1; i < _rates.length; i++) {
        require(_rates[i - 1] <= _rates[i], "PoolSetting/invalid-rate");
        require(_utils[i - 1] < _utils[i], "PoolSetting/invalid-util");
    }
    delete rates;
    delete utils;
    for (uint256 i = 0; i < _rates.length; i++) {
        rates[i] = _rates[i];
        utils[i] = _utils[i];
    }
    targetSupply = _targetSupply;
    utilsLen = _utils.length;

    emit SetBorrowInterestParams(msg.sender, _rates, _utils, targetSupply);
}
```

Listing 36.2 The improved *setBorrowInterestParams* function of the *PoolSetting* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 37	Recommended Removing Unused Code		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*Several files*		
Locations	*Several functions throughout multiple contracts*		

Detailed Issue

We found that unused codes can be removed for readability and maintainability as listed below.

Smart Contracts

- The *Manager* contract (*contracts/src/etc/Manager.sol*)

Functions

- The *_getNextLendingInterest* function of the *PoolBaseFunc* contract (L30 - 53)
- The *_loanLiquidationSwap* function of the *CoreBaseFunc* contract (L145 - 163)
- The *_validatePriceImpact* function of the *CoreSwapping* contract (L378 - 403)
- The *pause* function of the *FeeVault* contract (L101 - 104)
- The *unPause* function of the *FeeVault* contract (L106 - 109)

Imported Libraries

- The imported *SelectorPausable* of the *InterestVault* contract (L11)

Events

- The *SetCoreBorrowingAddress* of the *ILogicStorage* interface (L58)

Recommendations

We recommend removing the unused codes to improve the readability and maintainability of the protocol.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 38	Misspelled Variable And Parameter Names		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/interfaces/IFeeVault.sol* *contracts/src/factory/FwxFactory.sol* *contracts/src/core/logic/CoreSwapping.sol*		
Locations	*IFeeVault.sol L: 34* *FwxFactory.sol L: 59* *CoreSwapping.sol L: 542*		

Detailed Issue

We found that the following contracts contain spelling errors, which may confuse developers.

- The ***acution*** parameter (L34) in the *IFeeVault* interface
- The ***oldFwxFactort*** variable (L59) in the *FwxFactory* contract
- The ***reseveAmounts*** variable (L542) in the *CoreSwapping* contract

Recommendations

We recommend correcting spelling errors to enhance clarity and prevent potential confusion.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 39	Recommended Improving Comments To Reflect The Code		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/utils/PriceFeed.sol* *contracts/src/factory/logic/FwxFactoryValidator.sol* *contracts/src/core/logic/CoreSwapping.sol* *contracts/src/core/logic/CoreFutureOpening.sol*		
Locations	*PriceFeed.queryReturn L: 50* *PriceFeed._queryRateUSD L: 153* *FwxFactoryValidator._validatePair L: 199* *CoreSwapping._queryRate L: 511* *CoreFutureOpening._updateWalletAndValidateMarginForOpeningPosition L: 294*		

Detailed Issue

We have identified that some comments do not reflect the code. This issue could affect the transparency of the protocol. **For example, in the *PriceFeeds* contract, the comment at line 50 in code snippet 39.1 suggests the function returns zero during a pause, but the actual code at line 153 in code snippet 39.2 reverts during a pause.** Additionally, there are more misaligned comments in other contracts, for example:

1. Line 199 in the *FwxFactoryValidator* contract, where a todo comment is present even though the code is already implemented.
2. Line 511 in the *CoreSwapping* contract, where the comment mentions a precision mismatch as an example of the return value.
3. Line 294 in the *CoreFutureOpening* contract, where the comment indicates the use of the number one router, but the implemented code uses the number zero router.

PriceFeed.sol

```
50  //// NOTE: This function returns 0 during a pause, rather than a revert. Ensure
    calling contracts handle correctly. ///
51  function queryReturn(
52      address sourceToken,
53      address destToken,
54      uint256 sourceAmount
55  ) public view returns (uint256 destAmount) {
56      (uint256 rate, uint256 precision) = _queryRate(sourceToken, destToken);
57      destAmount = (sourceAmount * rate) / precision;
}
```

Listing 39.1 The comment states returning zero during a pause

PriceFeed.sol

```solidity
function _queryRateUSD(address token) internal view returns (uint256 rate, uint256 precision) {
    require(!globalPricingPaused, "PriceFeed/pricing-is-paused");
    require(pricesFeeds[token] != address(0), "PriceFeed/unsupported-address");
    AggregatorV2V3Interface feed = AggregatorV2V3Interface(pricesFeeds[token]);
    (, int256 answer, , uint256 updatedAt, ) = feed.latestRoundData();
    rate = uint256(answer);
    uint256 decimal = feed.decimals();

    rate = (rate * WEI_PRECISION) / (10 ** decimal);
    precision = WEI_PRECISION;

    require(block.timestamp - updatedAt < stalePeriod[token], "PriceFeed/price-is-stale");
}
```

Listing 39.2 The implemented code reverts during a pause

Recommendations

We recommend the team align the comments with the code by making necessary modifications. However, the team should ensure that both the code and comments align consistently with the business strategy outlined in the protocol.

Reassessment

The *FWX* team adopted our recommended code to address this issue by updating the logic to ensure alignment between the code and comments, consistent with the protocol's business strategy.

No. 40	Incorrectly Emitted Event Value		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/logicstorage/LogicStorage.sol* *contracts/src/core/logic/CoreSetting.sol*		
Locations	*LogicStorage.setAPHPoolAddress L: 125 - 133* *CoreSetting.setRouterAddresses L: 195 - 205*		

Detailed Issue

We found that the incorrect event emissions as shown in the listed below

- The *setAPHPoolAddress* function incorrectly uses the *_aphCoreAddress* state (L128 in code snippet 40.1) to log the old *APHPool* address (L131 in code snippet 40.1).

- The *setRouterAddresses* of the *CoreSetting* contract can emit the empty array of *routers* (L198 in code snippet 40.2) in case of inputting the empty *_routers* parameter (L195 in code snippet 40.2).

LogicStorage.sol

```
125 function setAPHPoolAddress(
126     address _address
127 ) external onlyAddressTimelockManager returns (bool) {
128     address oldAddress = _aphCoreAddress;
129     _aphPoolAddress = _address;
130
131     emit SetAPHPoolAddress(msg.sender, oldAddress, _address);
132     return true;
133 }
```

Listing 40.1 The *setAPHPoolAddress* function of the *LogicStorage* contract

CoreSetting.sol

```solidity
function setRouterAddresses(address[] memory _routers) external onlyAddressTimelockManager {
    require(_routers.length <= 5, "CoreSetting/router-addresses-beyond-limit-of-5");

    address[5] memory oldRouters = routers;
    for (uint16 i = 0; i < _routers.length; i++) {
        require(_routers[i] != address(0), "CoreSetting/router-address-is-zero");
        routers[i] = _routers[i];
    }

    emit SetRoutersAddress(msg.sender, oldRouters, routers);
}
```

Listing 40.2 The *setRouterAddresses* function of the *CoreSetting* contract

Recommendations

We recommend revising the mentioned incorrect event emission to improve the transparency and traceability of the protocol.

Reassessment

The *FWX* team has revised the event emission as shown in the Listing 40.3, 40.4 and 40.5.

LogicStorage.sol

```solidity
function setAPHPoolAddress(
    address _address
) external onlyAddressTimelockManager returns (bool) {
    address oldAddress = _aphPoolAddress;
    _aphPoolAddress = _address;

    emit SetAPHPoolAddress(msg.sender, oldAddress, _address);
    return true;
}
```

Listing 40.3 The *setAPHPoolAddress* function of the *LogicStorage* contract

CoreSetting.sol

```
function setRouterAddresses(address[] memory _routers) external
onlyAddressTimelockManager {
    require(_routers.length <= 5,
"CoreSetting/router-addresses-beyond-limit-of-5");
    require(_routers.length > 0, "CoreSetting/empty-routers");

    address[5] memory oldRouters = routers;
    delete routers;
    for (uint16 i = 0; i < _routers.length; i++) {
        require(_routers[i] != address(0),
"CoreSetting/router-address-is-zero");
        routers[i] = _routers[i];
    }

    emit SetRoutersAddress(msg.sender, oldRouters, routers);
}
```

Listing 40.4 The *setRouterAddresses* function of the *CoreSetting* contract

FWXFactorySetting.sol

```
function setWethHandlerAddress(address _wethHandler) external
onlyAddressTimelockManager {
    address oldWethHandler = wethHandler;
    wethHandler = _wethHandler;

    emit SetWethHandlerAddress(msg.sender, oldWethHandler, wethHandler);
}
```

Listing 40.5 The *setWethHandlerAddress* function of the *FWXFactorySetting* contract

No. 41	Enhancing Library Compatibility With Non-upgradeable Contracts		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/logicstorage/LogicStorageBase.sol* *contracts/src/core/FeeVault.sol*		
Locations	*Imported contracts*		

Detailed Issue

We found that **the *LogicStorageBase* and *FeeVault* contracts are non-upgradeable contracts that use the upgradeable contract and library** (The ***ManagerTimelockUpgradeable*** contract and ***MathUpgradeable*** library, respectively). We encourage utilizing the non-upgradeable library for consistency.

However, many libraries in the upgradeable contracts are derived from the non-upgradeable contracts and remain in the same code to enhance usage convenience when the contract is upgradeable.

Moreover, the *MathUpgradeable* library from the ***openzeppelin-contracts-upgradeable*** **has been removed** starting from version >= v5.0.0, forcing development to use the non-upgradeable version to avoid development confusion.

Recommendations

We recommend utilizing the non-upgradeable library for consistency.

Reassessment

The *FWX* team adopted our recommended code to fix this issue by using a non-upgradeable library.

No. 42	Recommended Improving The Error Messages		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/etc/ManagerTimelock.sol* *contracts/src/etc/ManagerTimelockUpgradeable.sol*		
Locations	*ManagerTimelock._onlyNoTimelockManager L: 39 - 41* *ManagerTimelock._onlyConfigTimelockManager L: 43 - 45* *ManagerTimelock._onlyAddressTimelockManager L: 47 - 49* *ManagerTimelockUpgradeable._onlyNoTimelockManager L: 46 - 48* *ManagerTimelockUpgradeable._onlyConfigTimelockManager L: 50 - 52* *ManagerTimelockUpgradeable._onlyAddressTimelockManager L: 54 - 56*		

Detailed Issue

We have identified some unclear error messages. For instance, **all the error messages related to unauthorized calls in the *ManagerTimelock* are the same, as "Manager/caller-is-not-the-manager"** as shown in the code snippet below.

This issue also happens in similar functions in the *ManagerTimelockUpgradeable* contract. These unclear error messages could make it harder to debug errors.

ManagerTimelock.sol

```
39  function _onlyNoTimelockManager() internal view {
40      require(noTimelockManager == msg.sender,
    "Manager/caller-is-not-the-manager");
41  }
42
43  function _onlyConfigTimelockManager() internal view {
44      require(configTimelockManager == msg.sender,
    "Manager/caller-is-not-the-manager");
45  }
46
47  function _onlyAddressTimelockManager() internal view {
48      require(addressTimelockManager == msg.sender,
    "Manager/caller-is-not-the-manager");
49  }
```

Listing 42.1 The unclear error messages of the *ManagerTimelock* contract

Recommendations

We advise the team to modify the error messages to be more detailed for each specific error. For instance, consider the following code recommendations.

However, it is crucial for the team to ensure that these updated error messages align with the protocol governance policies.

ManagerTimelock.sol

```
function _onlyNoTimelockManager() internal view {
    require(noTimelockManager == msg.sender,
"Manager/caller-is-not-the-no-timelock-manager");
}

function _onlyConfigTimelockManager() internal view {
    require(configTimelockManager == msg.sender,
"Manager/caller-is-not-the-config-timelock-manager");
}

function _onlyAddressTimelockManager() internal view {
    require(addressTimelockManager == msg.sender,
"Manager/caller-is-not-the-address-timelock-manager");
}
```

Listing 42.2 The code recommendation for specific error messages

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 43	Incorrect Filename		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/etc/SelectorPauseableUpgradeable.sol*		
Locations	-		

Detailed Issue

We have spotted a typo in the filename of the ***SelectorPausableUpgradeable*** contract. The current file name is ***SelectorPauseableUpgradeable.sol*** which is grammatically incorrect as shown in the code snippet below.

Recommendations

We advise the team to modify the filename to be grammatically correct and match the contract name as *SelectorPausableUpgradeable.sol.*

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 44	Unnecessary Data Overriding with _delegateCall		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/factory/proxy/FwxFactorySettingProxy.sol* *contracts/src/factory/proxy/FwxFactoryValidatorProxy.sol* *contracts/src/core/proxy/APHCoreProxy.sol* *contracts/src/core/proxy/APHCoreSettingProxy.sol*		
Locations	*Several functions throughout multiple contracts*		

Detailed Issue

We found that some of the functions implemented in the proxy contracts always retrieve the return data from the ***_delegateCall*** function to the implementation contracts, and some of the functions in those implementation contracts have no return value.

FwxFactorySettingProxy.sol

```
function setProxyAdmin(address _proxyAdmin) external override {
    bytes memory data = abi.encodeWithSelector(
        IFwxFactorySetting.setProxyAdmin.selector,
        _proxyAdmin
    );
    data = _delegatecall(fwxFactorySetting, data);
}
```

Listing 44.1 The *setProxyAdmin* function of the *FwxFactorySettingProxy* contract

Recommendations

We recommend revising the functions implemented in the proxy contracts to ensure consistency in handling return data from the ***_delegateCall*** function to the implementation contracts.

Reassessment

The *FWX* team has implemented our recommended code solution to resolve this issue by appropriately handling return data from the ***_delegateCall*** function.

No. 45	Inconsistency In Burnable Amount Logic		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/PoolToken.sol*		
Locations	*PoolToken._burnPToken L: 115 - 125*		

Detailed Issue

We found inconsistency in the implementation of the burn token logic within the *_burnPToken* function of the *PoolToken* contract (L115 - 125 in the code snippet below) compared to other instances of burn token logic.

PoolToken.sol

```
115  function _burnPToken(
116      address burner,
117      uint256 nftId,
118      uint256 burnAmount
119  ) internal returns (uint256) {
120      pTokenTotalSupply -= burnAmount;
121      tokenHolders[nftId].pToken -= burnAmount;
122
123      emit BurnPToken(burner, nftId, burnAmount);
124      return burnAmount;
125  }
126
127  function _burnAtpToken(
128      address burner,
129      uint256 nftId,
130      uint256 burnAmount,
131      uint256 price
132  ) internal returns (uint256) {
133      burnAmount = MathUpgradeable.min(burnAmount, tokenHolders[nftId].atpToken);
134
135      atpTokenTotalSupply -= burnAmount;
136      tokenHolders[nftId].atpToken -= burnAmount;
137
138      emit BurnAtpToken(burner, nftId, burnAmount, price);
139      return burnAmount;
140  }
141
```

```
function _burnItpToken(
    address burner,
    uint256 nftId,
    uint256 burnAmount,
    uint256 price
) internal returns (uint256) {
    burnAmount = MathUpgradeable.min(burnAmount, tokenHolders[nftId].itpToken);

    itpTokenTotalSupply -= burnAmount;
    tokenHolders[nftId].itpToken -= burnAmount;

    emit BurnItpToken(burner, nftId, burnAmount, price);
    return burnAmount;
}
```

Listing 45.1 The inconsistency implementation of the burn token logic within the *_burnPToken* function

Recommendations

We recommend applying a **boundary check** in the *_burnPToken* function for better consistency compared to other burn token logic. Furthermore, it will help to prevent potential cases of arithmetic underflow reverts.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

PoolToken.sol

```
// (...SNIPPED...)

function _burnPToken(
        address burner,
        uint256 nftId,
        uint256 burnAmount
    ) internal returns (uint256) {
        burnAmount = MathUpgradeable.min(burnAmount, tokenHolders[nftId].pToken);
        pTokenTotalSupply -= burnAmount;
        tokenHolders[nftId].pToken -= burnAmount;

        emit BurnPToken(burner, nftId, burnAmount);
        return burnAmount;
    }
```

Listing 45.1 The improved *_burnPToken* function of the *PoolToken* contract

No. 46	Deposit Native Token Failure Due To Requirement Conflict		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/pool/logic/PoolLending.sol*		
Locations	*PoolLending.depositFor L: 295 - 337*		

Detailed Issue

We identified a discrepancy in the *depositFor* function which is marked as payable, indicating it should accept native token deposits. However, a conflicting requirement *(require(msg.value == 0, "PoolLending/unsupported-native-token");)* on line 313 prevents the receipt of any native tokens. This inconsistency renders the function incapable of accepting native token transactions as intended.

PoolLending.sol

```
295  function depositFor(
296      address caller,
297      uint256 nftId,
298      uint256 depositAmount,
299      bytes calldata data
300  )
301      external
302      payable
303      nonReentrant
304      whenFuncNotPaused(msg.sig)
305      returns (uint256 mintedP, uint256 mintedAtp, uint256 mintedItp)
306  {
307      /**
308          * NOTE
309          *  caller      = user
310          *  msg.sender  = FwxFactory
311          */
312
313      require(msg.value == 0, "PoolLending/unsupported-native-token");
314      require(
315          caller == IMembership(membershipAddress).ownerOf(nftId),
316          "PoolLending/deposit-for-unowned-nft"
317      );
```

Listing 46.1 The *depositFor* function of the *PoolLending* contract

Recommendations

As a requirement conflict of the *depositFor* function, we recommend the team ensure the behavior of this function that is supposed to receive the native tokens or not.

Reassessment

The *FWX* team fixed this issue by removing the ***payable*** modifier of the ***depositFor*** function to consistent the business requirement and protocol's functionality.

No. 47	Inability To Disable The Routers After Being Set		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/logic/CoreSetting.sol*		
Locations	*CoreSetting.setRouterAddresses L: 195 - 205*		

Detailed Issue

The *setRouterAddresses* function is responsible for setting the router addresses of the protocol. However, we have identified certain limitations in its functionality. To elaborate, **it is not possible to disable routers without substituting them with alternative router addresses**. Consider the following scenario

1. The current routers are *[0x01, 0x02, 0x03]*
2. The administrator decides to disable the *0x03* router, so he invokes the function with *[0x01, 0x02, 0x00]*
3. Unfortunately, the function reverts due to the absence of a zero address validation mechanism, as shown in the code snippet below

CoreSetting.sol

```
195 function setRouterAddresses(address[] memory _routers) external onlyAddressTimelockManager {
196      require(_routers.length <= 5, "CoreSetting/router-addresses-beyond-limit-of-5");
197
198      address[5] memory oldRouters = routers;
199      for (uint16 i = 0; i < _routers.length; i++) {
200          require(_routers[i] != address(0), "CoreSetting/router-address-is-zero");
201              routers[i] = _routers[i];
202      }
203
204      emit SetRoutersAddress(msg.sender, oldRouters, routers);
205 }
```

Listing 47.1 The *setRouterAddresses* function of the *CoreSetting* contract

Furthermore, if the administrator attempts to invoke the function with *[0x01, 0x02]* instead, the routers remain unchanged at *[0x01, 0x02, 0x03]* since the index 3 of the array is not reassigned.

Recommendations

We recommend the team modify the *setRouterAddresses* function to reset the *routers* state to be empty before re-assigning its value. Additionally, we recommend adding validation to prevent the *routers* state from being empty.

CoreSetting.sol

```
function setRouterAddresses(address[] memory _routers) external onlyAddressTimelockManager {
     require(_routers.length <= 5, "CoreSetting/router-addresses-beyond-limit-of-5");
     require(_routers.length > 0, "CoreSetting/empty-router-addresses");

     address[5] memory oldRouters = routers;
     delete routers;
     for (uint16 i = 0; i < _routers.length; i++) {
         require(_routers[i] != address(0), "CoreSetting/router-address-is-zero");
             routers[i] = _routers[i];
     }

     emit SetRoutersAddress(msg.sender, oldRouters, routers);
}
```

Listing 47.2 The improved *setRouterAddresses* function of the *CoreSetting* contract

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

No. 48	Mismatched NFT Owner Event Emission		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Fixed
Associated Files	*contracts/src/core/logic/CoreFutureOpening.sol* *contracts/src/core/logic/CoreFutureClosing.sol*		
Locations	*CoreFutureOpening._openPosition L: 25 - 172* *CoreFutureClosing._closePosition L: 24 - 99* *CoreFutureClosing._liquidatePosition L: 106 - 230*		

Detailed Issue

We found the incorrect NFT owner value of the event emissions listed below

- The *TransferForwTradingReward* emission of the *CoreFutureOpening* contract (L94)
- The *ClosePosition* emission of the *CoreFutureClosing* contract (L82)
- The *ClosePosition* emission of the *CoreFutureClosing* contract (L141)

Use the *TransferForwTradingReward* emission of the *CoreFutureOpening* as an example to elaborate, the *msg.sender* at line 94 in the code snippet below does not guarantee they are the *NFT* owner (EOA) cause the *APHPool* (Smart Contract) could call the *openPosition* function as well.

This incorrect event emission may affect the transparency and traceability of the protocol.

CoreFutureOpening.sol

```
25  function _openPosition(
26      APHLibrary.OpenPositionParams memory params,
27      APHLibrary.TokenAddressParams memory addressParams
28  ) internal {

        // (...SNIPPED...)

86              if (forwAmount != 0) {
87                  _transferFromOut(
88                      forwTradingVaultAddress,
89                      _getTokenOwnership(params.nftId),
90                      forwAddress,
91                      forwAmount
92                  );
93                  emit TransferForwTradingReward(
 94                     msg.sender,
 95                     params.nftId,
 96                     tmp.pairByte,
 97                     forwAmount
 98                 );
 99             }
100         }
101     }

    // (...SNIPPED...)
```

Listing 48.1 The example one of the incorrect NFT owner value for the event emissions

Recommendations

We recommend revising the mentioned incorrect event emission by getting the actual *NFT* owner with the *_getTokenOwnership* function (L94) instead as shown in the code snippet below.

Please apply the recommendation to the event emissions listed below

- **The *TransferForwTradingReward* emission of the *CoreFutureOpening* contract (L94)**
- **The *ClosePosition* emission of the *CoreFutureClosing* contract (L82)**
- **The *ClosePosition* emission of the *CoreFutureClosing* contract (L141)**

CoreFutureOpening.sol

```
 25 function _openPosition(
 26     APHLibrary.OpenPositionParams memory params,
 27     APHLibrary.TokenAddressParams memory addressParams
 28 ) internal {

    // (...SNIPPED...)

 86             if (forwAmount != 0) {
 87                 _transferFromOut(
 88                     forwTradingVaultAddress,
 89                     _getTokenOwnership(params.nftId),
 90                     forwAddress,
 91                     forwAmount
 92                 );
 93                 emit TransferForwTradingReward(
 94                     _getTokenOwnership(msg.sender),
 95                     params.nftId,
 96                     tmp.pairByte,
 97                     forwAmount
 98                 );
 99             }
100         }
```

```
    }

    // (...SNIPPED...)
```

Listing 48.2 The example of getting the actual NFT owner for the event emission

The recommended code provides the concept of how to remediate this issue only. The code should be adjusted accordingly.

Reassessment

The *FWX* team adopted our recommended code to fix this issue.

CoreFutureOpening.sol

```
// (...SNIPPED...)

if (forwAmount != 0) {
    address nftOwner = _getTokenOwnership(params.nftId);
    _transferFromOut(forwTradingVaultAddress, nftOwner, forwAddress,
forwAmount);
    emit TransferForwTradingReward(
        nftOwner,
        params.nftId,
        tmp.pairByte,
        forwAmount
    );
}
```

Listing 48.3 The improved *_openPosition* function of the *CoreFutureOpening* contract

CoreFutureClosing.sol

```
// (...SNIPPED...)

address nftOwner = _getTokenOwnership(nftId);
emit ClosePosition(
    nftOwner,
    nftId,
    _posId,
    params.closingSize,
    result.rate,
    result.pnl,
    posState.isLong,
    !posState.active,
    posState.pairByte,
    result.collateralSwappedAmountReturn,
    result.router
);

emit CollectFees(
    nftOwner,
    nftId,
    pos.id,
    posState.pairByte,
    uint128(result.tradingFee),
    uint128(result.swapFee),
    uint128(result.interestPaid),
    0,
    0,
    0
);
```

Listing 48.4 The improved *_closePosition* function of the *CoreFutureClosing* contract

CoreFutureClosing.sol

```
tmp.nftOwner = _getTokenOwnership(nftId);
{
    tmp.closingSize = posState.isLong ? pos.contractSize : pos.borrowAmount;
    APHLibrary.ClosePositionParams memory params =
APHLibrary.ClosePositionParams(
        nftId,
        pairByte,
        tmp.posId,
        tmp.closingSize,
        _getNFTRankInfo(nftId).tradingFee,
        true
    );

    result = posState.isLong ? _closeLong(params) : _closeShort(params);

    emit ClosePosition(
        tmp.nftOwner,
        nftId,
        tmp.posId,
        params.closingSize,
        result.rate,
        result.pnl,
        posState.isLong,
        false,
        posState.pairByte,
        result.collateralSwappedAmountReturn,
        result.router
    );
}
```

Listing 48.5 The improved *_liquidatePosition* function of the *CoreFutureClosing* contract

No. 49	Price Impact Due To Low Liquidity: DEX vs Oracle Price Discrepancy		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/logic/CoreSwapping.sol*		
Locations	*CoreSwapping._getAmountsWithRouterSelection L: 244 - 273*		

Detailed Issue

The hedging protocol coordinately uses the *Decentralized Exchange (DEX)* and the *Oracle Price Feed* to prevent the risk of the single source price impact as shown in the code snippet below. **To elaborate, the invoking of *_checkPriceDiff* function (L269) will check the price between *Oracle Price Feed* and *DEX* should not exceed the configured *maxOraclePriceDiffPercent*** first before allowing it to open, close, and liquidate position.

However, the price of the *Decentralized Exchange (DEX)* fluctuation relies on the liquidity and swap amount, from this point may create risks for the use.

Please consider a scenario that the low liquidity *DEX* may impact the *price diff* exceeding the *maxOraclePriceDiffPercent* (L269) that prevents performing the open, close, and liquidate position operation, in particular, liquidate must be done in time to reduce the risk of pool loss.

CoreFutureOpening.sol

```
244 function _getAmountsWithRouterSelection(
245     bool isExactOutput,
246     bytes32 pairByte,
247     uint256 amountInput,
248     address[] memory path,
249     uint256 expectedRate,
250     uint256 slippage
251 ) internal view returns (uint256[] memory amounts, uint256 swapFee, uint256
    routerIndex) {
252     routerIndex = 0;
253     Rates memory rates;
254     SwapConfig memory cfg = swapConfigs[routers[routerIndex]][pairByte];
255 
256     // verifying for external dex
257     if (!_isRouterUsable(routerIndex, isExactOutput, amountInput, path))
```

```
        revert("CoreSwapping/cannot-find-usable-router");

    (amounts, swapFee) = _getAmounts(isExactOutput, true, routerIndex, amountInput, path);
    rates.swapRate = _calculateSwapRate(pairByte, path, amounts);
    if (slippage != 0 && !_checkPriceDiff(expectedRate, rates.swapRate, slippage))
        revert("CoreSwapping/slippage-too-low");

    rates.oracleRate = _queryOraclePrice(pairByte);
    rates.reserveRate = _getReserveRate(pairByte, routerIndex, path);
    if (
        rates.oracleRate != 0 &&
        !_checkPriceDiff(rates.oracleRate, rates.reserveRate, cfg.maxOraclePriceDiffPercent)
    ) revert("CoreSwapping/price-diff-oracle-exceed");

    return (amounts, swapFee, routerIndex);
}
```

Listing 49.1 The _getAmountsWithRouterSelection function of the CoreFutureOpening contract

Recommendations

There is no recommendation code for this issue as it might break the contract functionality and require a decision from the *FWX* team in terms of business and protocol's core functionality.

However, we recommend the *FWX* team adjust the *maxOraclePriceDiffPercent* to suit the situation. Additionally, we advise that the team revise the protocol to support multiple *Decentralized Exchange (DEX)* to further reduce the price impact and risk of the single source.

Reassessment

The *FWX* statement has acknowledged with the statement:

"This is an acknowledged issue. We recognize that price manipulation is a dangerous attack vector in blockchain (and de-fi projects), but Permissionless markets are designed to be more flexible and decentralized than the official market. The solution is that we could provide the information or risk assessment on our website."

No. 50	Recommended Enforcing Checks-Effects-Interactions Pattern		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/logic/CoreFutureClosing.sol* *contracts/src/core/logic/CoreFutureBaseFunc.sol* *contracts/src/core/logic/CoreBaseFunc.sol*		
Locations	*CoreFutureClosing._liquidatePosition L: 106 - 230* *CoreFutureClosing._resetPosition L: 523 - 526* *CoreFutureBaseFunc._getPositionMargin L: 178 - 224* *CoreBaseFunc._getRankInfo L: 56 - 60*		

Detailed Issue

We found that some functions do not follow the checks-effects-interactions pattern which is the best practice for developing secure smart contracts. To elaborate, consider this process

1. The *_liquidatePosition* function of the *CoreFutureClosing* contract first calls the *_getPositionMargin* of the *CoreFutureBaseFunc* contract. This *_getPositionMargin* function checks various states, such as *positionStates*, used in the checking mechanism (L 189 - 190 in the code snippet 50.1). This is the check step.
2. The *CoreFutureBaseFunc* contract then calls the *_getNFTRankInfo* function, resulting in an external call (L 59 in the code snippet 50.2). This is the interaction step.
3. In the later part of the flow, the *_resetPosition* function is called, updating the *positionStates* state (L 524 in the code snippet 50.3). This is the effect step.

Although this scenario is not vulnerable to reentrancy attacks, we still recommend the FWX team use the checks-effects-interactions pattern in all functions. Following this pattern is considered a best practice for developing secure smart contracts.

CoreFutureBaseFunc.sol

```
178 function _getPositionMargin(
179     uint256 nftId,
180     bytes32 pairByte,
181     bool checkPriceDiff,
182     bool isLiquidate
183 ) internal returns (uint256 margin) {
184     GetPositionMarginTmpStruct memory tmp;
```

```
    Pair memory pair = pairs[pairByte];
    Position memory pos = positions[nftId][pairByte];
    uint256 collateralPrecision = tokenPrecisionUnit[pair.pair0];
    uint256 underlyingPrecision = tokenPrecisionUnit[pair.pair1];
    PositionState memory posState = positionStates[nftId][pos.id];
    require(pos.id != 0 || posState.active, "CoreTrading/position-is-not-active");

    // (...SNIPPED...)
```

Listing 50.1 The check step in the *CoreFutureBaseFunc* contract

CoreBaseFunc.sol

```
function _getRankInfo(
    uint8 rank
) internal view returns (StakePoolBase.RankInfo memory rankInfo) {
    rankInfo = IStakePool(IMembership(membershipAddress).currentPool()).rankInfos(rank);
}
```

Listing 50.2 The interaction step in the *CoreBaseFunc* contract

CoreFutureClosing.sol

```
function _resetPosition(uint256 nftId, uint256 posId, bytes32 pairByte) private {
    positionStates[nftId][posId].active = false;
    positions[nftId][pairByte] = positions[0][0];
}
```

Listing 50.3 The effect step in the *CoreFutureClosing* contract

Recommendations

We recommend the *FWX* team to change the code pattern from checks-interactions-effects to the checks-effects-interactions pattern at multiple places where it occurs. However, we require the *FWX* team to decide on the modifications as it might break the contract functionality.

Reassessment

The *FWX* has acknowledged this issue with the statement:

"It works as design. For example, in this issue, this flow involves two features: future trading and staking.

- *check step: check states of the position and positionState, which involve future trading features.*
- *interaction step: interact with staking benefits, which involve staking features.*
- *effects step: update the position and positionState, which involve future trading features.*

Future trading does not affect any staking features.

If future trading does not allow users to open or close positions using smart contracts, then we assume that attacking flaws between two features in the same transaction is impossible, e.g., stake to get the highest benefits, then close position, and unstake to get their token back to close with the highest benefits."

Detailed Issue From The Reassessment Process

This section provides all issues that we found from the reassessment process.

No. 1	Lack Of Price Slippage Control Mechanism		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/core/logic/CoreFutureWallet.sol* *contracts/src/core/logic/CoreFutureBaseFunc.sol* *contracts/src/core/logic/CoreFutureClosing.sol*		
Locations	*CoreFutureClosing._closeLong L: 54 - 346* *CoreFutureClosing._closeShort L: 348 - 524* *CoreFutureBaseFunc._getUnrealizedPNL L: 112 - 178* *CoreSwapping.positionLiquidationSwap L: 96 - 126*		

Detailed Issue

The hedging protocol coordinately uses the ***Decentralized Exchange (DEX)*** and the ***Oracle Price Feed*** to prevent the risk of the single source price impact. **For example, the invoking of *_checkPriceDiff* function (L122) will check the price between *Oracle Price Feed* and *DEX* should not exceed the configured price diff percent** first before allowing it to liquidate position as shown in the code snippet below.

CoreSwapping.sol

```
 96 function positionLiquidationSwap(
 97     bool isExactOutput,
 98     bytes32 pairByte,
 99     uint256 amountIn,
100     uint256 amountOut,
101     address[] memory path,
102     address receiver
103 ) external returns (uint256[] memory amounts, uint256 swapFee, address router) {
104     uint256 routerIndex = 0; // external dex
105     uint256 oracleRate = _queryOraclePrice(pairByte);
106 
107     // get actual rate from external dex
108     router = routers[routerIndex];
109     (amounts, swapFee) = _getAmounts(
110         isExactOutput,
111         true,
112             routerIndex,
113             isExactOutput ? amountOut : amountIn,
114             path
115         );
116         uint256 swapRate = _calculateSwapRate(pairByte, path, amounts);
117 
118         // compare actual rate to oracle rate
119         SwapConfig memory cfg = swapConfigs[router][pairByte];
120         require(
121             oracleRate == 0 ||
122                 _checkPriceDiff(oracleRate, swapRate, cfg.maxLiquidationOraclePriceDiffPercent),
123             "CoreSwapping/liquidate-price-diff-oracle-exceed"
124         );
125         _swap(isExactOutput, routerIndex, amountIn, amountOut, path, receiver);
126 }
```

Listing 1.1 The example *positionLiquidationSwap* function of the *CoreSwapping* contract that invokes the *_checkPriceDiff* function

However, the ***_checkPriceDiff*** invoking can bypass the price slippage check when the *oracleRate* returns 0 (L122 in the code snippet above) from the ***pricesFeeds[token]*** is ***zero address*** or ***globalPricingPaused*** is ***false*** (L157 in the code snippet below).

When the token lacks Oracle price feeds, there is no slippage in the following processes, thereby increasing the risk of price manipulation in listing functions:

- The *withdrawCollateral* function from the *CoreFutureWallet* contract.
- The *closePosition* function from the *CoreFutureClosing* contract.
- The *liquidatePosition* function from the *CoreFutureClosing* contract.

PriceFeed.sol

```
156 function _queryRateUSD(address token) internal view returns (uint256 rate, uint256 precision) {
157     if (pricesFeeds[token] == address(0) || globalPricingPaused) return (0, 0);

        // (...SNIPPED...)
```

Listing 1.2 The *_queryRateUSD* function of the *PriceFeeds* contract

Recommendations

We recommend implementing slippage control measures for each of the mentioned processes.

This could involve introducing checks or safeguards to ensure that prices are not manipulated during activities such as **withdrawing collateral**, **closing positions**, or **liquidating positions.**

Reassessment

The *FWX* statement has acknowledged with the statement:

“This is an acknowledged issue. We recognize that price manipulation is a dangerous attack vector in blockchain (and de-fi projects), but Permissionless markets are designed to be more flexible and decentralized than the official market. The solution is that we could provide the information or risk assessment on our website.”

No. 2	Lack Of Lender Loss Tracking		
Risk	High	Likelihood	Medium
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/pool/logic/PoolLending.sol*		
Locations	*PoolLending._withdraw L: 183 - 262*		

Detailed Issue

The future trading feature can result in losses by repaying less than the borrowed loan, thereby creating debt for the lenders in the Lending-Borrowing Pool during the closing and/or liquidation process.

CoreFutureClosing.sol

```
function _closeShort(
    APHLibrary.ClosePositionParams memory params
) internal returns (APHLibrary.ClosePositionResponse memory result) {

    // (...SNIPPED...)

    if (isCritical) {
        // ! LOSS
        // update pool stat
        poolStat.totalBorrowAmountFromTrading -= result.repayAmount;
        poolStat.borrowInterestOwedPerDayFromTrading -= pos.interestOwePerDay;

        IAPHPool(assetToPool[pair.pair1]).addLoss(result.repayAmount - tmp.actualCollateral);
        result.tradingFee = 0;
        result.pnl = APHLibrary._calculatePNL(
            pos.entryPrice,
            result.rate,
            result.repayAmount,
            underlyingPrecision
        );
```

Listing 2.1 The example adding *loss* amount to the *APHPool*, the *_closeShort* function of the *CoreFutureClosing* contract

Lenders are responsible to absorb those losses by including loss in calculating the actual principal withdrawal amount leading to decreasing the power to withdraw all their principal.

PoolLending.sol

```
function _withdraw(
    address receiver,
    uint256 nftId,
    uint256 withdrawAmount
) internal returns (WithdrawResult memory) {

    // (...SNIPPED...)

    uint256 lossBurnAmount = MathUpgradeable.min(withdrawAmount -
actualWithdrawAmount, loss);
    loss -= lossBurnAmount;


    // (...SNIPPED...)

}
```

Listing 2.3 The *_withdraw* function of the *PoolLending* contract that does not contain the loss tracking for each lender

However, there is no on-chain handling of losses for each lender, and the accumulated loss will be reset once the *APHPool* is empty, potentially exposing lenders in each permissionless market to the risk of bad debt from Future trading participants.

Recommendations

We recommend introducing on-chain loss tracking for each lender and implementing a mitigation process to remedy potential losses, thereby reducing the risk of bad debt.

Reassessment

The *FWX* statement has acknowledged with the statement:

"The FWX team has got an in-house off-chain service that stores and aggregates events emitted. In the event that they are going to remit losses to users on the permissionless, they could use stored historical data to calculate the remit amount per user."

No. 3	Potential Over-Distribution Of Lending Bonuses		
Risk	Medium	Likelihood	Low
		Impact	High
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/src/pool/logic/PoolLending.sol*		
Locations	*PoolLending._claimTokenInterest L: 264 - 302*		

Detailed Issue

From Issue **No. 27, Potentially Underflow Revert On Profit Distribution,** we found that the *bonusAmount* could possibly be greater than the left side of the *profitAmount* calculation on line 279. The FWX team has mitigated that potential underflow issue.

However, we discovered that the **over-distributed *bonusAmount*** value **is continually used in the claim interest process** through the *claimTokenInterest* function of the *PoolLending* contract.

As a result, the lender could potentially claim interest and receive a bonus greater than their profit distribution amount. The lender that claims lastly will be affected by receiving less than their profit.

PoolLending.sol

```
function claimTokenInterest(
     uint256 nftId,
     uint256 claimAmount
 ) external nonReentrant whenFuncNotPaused(msg.sig) returns (WithdrawResult memory result) {
     nftId = _getUsableToken(msg.sender, nftId);
     result = _claimTokenInterest(msg.sender, nftId, claimAmount);
     _transferFromOut(
         interestVaultAddress,
         msg.sender,
         tokenAddress,
         result.tokenInterest + result.tokenInterestBonus
     );
  return result;
}
```

Listing 3.1 The *claimTokenInterest* function of the *PoolLending* contract

Consider the following scenario:

- The **APH Pool has 2 shares of lenders with the same value of principle in the pool before interest is accrued**, meaning that both have the same power to claim interest.

0. Assume the initialized state for demonstration:

- All Interest occurs = _300 * 1e18_
- _heldTokenInterest = (300 * 1e18) * 10% = 30 * 1e18_
- _claimableInterest = 300 * 1e18 - heldTokenInterest = 270 * 1e18_
- _interestBonusLending_ = **_11.12%_**

Assume the power to claim of each lender (2 lenders) = 135 * 1e18

1. Lender A claims ALL their interest:

- _claimableAmount_ = 135 * _1e18_
- _bonusAmount_ = (135 * 1e18) * interestBonusLending% = **_15.012 * 1e18_**
- _profitAmount_ = (135 * 1e18) * 10 / (100 - 10) = **_15 * 1e18_**

The _bonusAmount_: _15.012 * 1e18_ is greater than _profitAmount_: _15 * 1e18_

Actual profitAmount = _15 * 1e18 - min(15.012 * 1e18, 15 * 1e18) = 0_

2. However, the value that passes to the _withdrawTokenInterest_ function still is:

IInterestVault(interestVaultAddress).withdrawTokenInterest(
claimable: claimableAmount,
bonus: 15.012 * 1e18, // the over value
profit: 0);

PoolLending.sol

```
function _claimTokenInterest(
    address receiver,
    uint256 nftId,
    uint256 claimAmount
) internal returns (WithdrawResult memory result) {
    uint256 itpPrice = _getInterestTokenPrice();
    PoolTokens storage tokenHolder = tokenHolders[nftId];

    uint256 claimableAmount;
    if (((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) >
tokenHolder.pToken) {
        claimableAmount =
            ((tokenHolder.itpToken * itpPrice) / PRECISION_UNIT) -
            tokenHolder.pToken;
    }

    claimAmount = MathUpgradeable.min(claimAmount, claimableAmount);

    uint256 burnAmount = _burnItpToken(
        receiver,
        nftId,
        (claimAmount * PRECISION_UNIT) / itpPrice,
        itpPrice
    );
    uint256 bonusAmount = (claimAmount * _getPoolRankInfo(nftId).interestBonusLending) /
        WEI_PERCENT_UNIT;

    uint256 feeSpread = IAPHCore(coreAddress).feeSpread();
    uint256 profitAmount = ((claimAmount * feeSpread) / (WEI_PERCENT_UNIT - feeSpread));
    profitAmount -= MathUpgradeable.min(bonusAmount, profitAmount);

    (claimAmount, bonusAmount, profitAmount) = IInterestVault(interestVaultAddress)
        .withdrawTokenInterest(claimAmount, bonusAmount, profitAmount);

    emit ClaimTokenInterest(receiver, nftId, claimAmount, bonusAmount, burnAmount);

    result.tokenInterest = claimAmount;
    result.itpTokenBurn = burnAmount;
    result.tokenInterestBonus = bonusAmount;
}
```

Listing 3.2 The *_claimTokenInterest* function of the *PoolLending* contract

3. As the ***bonusAmount***: ***15.012 * 1e18 < heldTokenInterest***: ***30 * 1e18*** and the *bonus* + *profit* amount is also less than the *heldTokenInterest*, **the bonus will be claimed as *15.012 * 1e18* and transferred back to the claimer**.

The remaining value of heldTokenInterest: *(30 - 15.012) * 1e18 = 14.988 * 1e18.*

In the scenario above, **when another lender claims all their interest, they will only receive *14.988 * 1e18* tokens of bonus and/or profit, while they both hold the same power of claim.**

Recommendations

We recommend implementing logic to ensure that the *bonusAmount* is appropriately bounded by the *profitAmount* to avoid situations where the bonus exceeds the profit.

Reassessment

The *FWX* statement has acknowledged with the statement:

“The FWX team has verified that the interestBonusLending will be below 11.11%. We recognize that certain lenders may not claim their entire interest bonus, leaving behind minimal amounts typically considered negligible.”

No. 4	Out Of Audit Scope		
Risk	Informational	Likelihood	Low
		Impact	Low
Functionality is in use	In use	Status	Acknowledged
Associated Files	*contracts/interfaces/IHelperFutureTradePermissionless.sol* *contracts/interfaces/IHelperPoolPermissionless.sol* *contracts/interfaces/IMarketIndexer.sol* *contracts/src/helper/HelperBase.sol* *contracts/src/helper/HelperFutureTradePermissionless.sol* *contracts/src/helper/HelperPoolPermissionless.sol* *contracts/src/helper/HelperUtils.sol* *contracts/src/helper/HelperUtilsFutureTrade.sol* *contracts/src/helper/MarketIndexer.sol*		
Locations	*Several functions throughout multiple contracts*		

Detailed Issue

The following listed interfaces and contracts below were added during the reassessment process.

- The *IHelperFutureTradePermissionless* interface
- The *IHelperPoolPermissionless* interface
- The *IMarketIndexer* interface
- The *HelperBase* contract
- The *HelperFutureTradePermissionless* contract
- The *HelperPoolPermissionless* contract
- The *HelperUtils* contract
- The *HelperUtilsFutureTrade* contract
- The *MarketIndexer* contract
- The *SetMarketIndexer* event in the *IFwxFactorySetting* interface
- The *CollectFees* event in the *CoreFutureTradingEvent* contract

Therefore, any use of these newly added interfaces, events, or functions within other contracts is not covered by this current audit and requires a full security review.

Recommendations

We recommend that the *FWX* team conducts a full security audit for the complete version of the interfaces and contracts listed above. This step is crucial to ensure the security of the contract.

Reassessment

The *FWX* team has acknowledged this issue.

Appendix

About Us

Founded in 2020, Valix Consulting is a blockchain and smart contract security firm offering a wide range of cybersecurity consulting services such as blockchain and smart contract security consulting, smart contract security review, and smart contract security audit.

Our team members are passionate cybersecurity professionals and researchers in the areas of private and public blockchain technology, smart contract, and decentralized application (DApp).

We provide a service for assessing and certifying the security of smart contracts. Our service also includes recommendations on smart contracts' security and gas optimization to bring the most benefit to users and platform creators.

Contact Information

✉ info@valix.io

https://www.facebook.com/ValixConsulting

https://twitter.com/ValixConsulting

https://medium.com/valixconsulting

References

Title	Link
OWASP Risk Rating Methodology	https://owasp.org/www-community/OWASP_Risk_Rating_Methodology
Smart Contract Weakness Classification and Test Cases	https://swcregistry.io/

ValiX